UNIVERSITY OF OSLO
Department of Informatics

False positive reduction through IDS network awareness

Jostein Haukeli

Network and System Administration

Oslo University College

May 23, 2012

False positive reduction through IDS network awareness

Jostein Haukeli

Network and System Administration
Oslo University College

May 23, 2012

Abstract

The common intrusion detection system is unable to determine the relevance of the alerts it generates because it lacks network and context awareness. A prototype was developed with the purpose of reducing the amount of false positives found in these systems. The prototype has the ability to determine the relevance of each alert by investigating the alert's vulnerability information and the target's host information. Challenges with passive fingerprinting of hosts behind Network Address Translation and in dynamic networks were also discussed and solved. Testing on real network traffic indicated that the prototype was successful in correctly categorizing a variety of alerts by assigning scores to each alert. This way the alerts can be ordered by their likeliness of being true positives, and the number of alerts that the system administrator has to investigate is reduced to a manageable size.

Contents

List of Figures

List of Tables

Code

Chapter 1

Introduction

1.1 Motivation

According to projections made by Cisco, Internet traffic will continue to rapidly increase over the next few years, estimating a growth by a factor of four in the period from 2010 to 2015 [1]. Furthermore Cisco predicts there will be twice as many networked devices as there are people on the planet by 2015, in contrast to one device per person in 2010 [1]. These developments are important points to be aware of in regards to Intrusion Detection Systems (IDS) and fighting malicious traffic on the Internet, because it gives us some idea of what challenges we will be facing. From an intrusion detection point of view increased Internet traffic means we will have to analyze a larger amount of data. In all likelihood we will also see an increased number of new services surfacing which are also suspicable to malicious usage. The most obvious ways to deal with these problems would perhaps be with higher performance hardware and software, while other approaches would be concerned with changing how the malicious traffic is identified or logs are analyzed.

In a threat report from 2010, the network security company Symantec revealed some disturbing trends, estimating a 93% increase in number of web attacks in the period from 2009 to 2010 [2]. As a result of the previously mentioned factors it is becomming increasingly difficult for system administrators to keep up with the ever increasing log sizes produced by IDSs. This problem is amplified further by the large number of false positives or otherwise irrelevant alerts usually produced by the common IDS. Studies have shown that over 90% of the alerts generated by some IDSs may be false positives [3, 4]. When a system administrator is faced with such large erroneous logs where investi-

gation must be done at a rapid pace to keep up with newly generated alerts, the risk of missing out on real security threats becomes problematic. The high orrucence of false postivives when investigating possible intrusions can also be demotivating and tedious for the administrator.

Network awareness in intrusion detection is not a recently proposed idea. Some IDS vendors such as Cisco with the no longer sold Cisco Threat Response 2.0 (CRT)[5] and Tenable Network Security with their Lighting Console 2.0 (product discontinued and product page removed) have provided solutions with the ability to scan and investigate the protected network, in order to verify the validity of alerts for almost 10 years. When these features were introduced they were widely considered to be somewhat unsophisticated, relying on time and resource heavy active scanning techniques to familiarize themselves with the network topology.

In later years, other IDS entrepreneurs have shown interest in the benefits their networks can gain from keeping track of the system under protection[6, 7]. More commercially available software products have since been implemented with improved network awareness capabilities. One example of this can be seen in the popular Sourcefire 3D solution which now boasts components providing Real-time User Awareness (RUA)[8] and Real-time Network Awareness (RNA)[9]. Using the information gathered by these additions, events can be further investigated automatically by the IDS to verify the likelihood of an attack being successful, as well as providing more verbose alerts of what users were involved and weaknesses were exploited.

Common for the existing solutions however is the fact that they are commercially available and closed source software. As a result, there is little information freely available on this topic. This makes it difficult for less resourceful competitors such as smaller groups of open source developers to stay competitive.

1.1.1 Problem statement

The main purpose of this thesis is to demonstrate a proof of concept addition to intrusion detection systems. This proof of concept will be demonstrated by implementing a prototype solution, introducing network and context awareness to IDS with the purpose of reducing the number of erroneous alerts.

The three following research questions will be answered in this project:

I. In what way can existing open source software tools best be used for collecting host information relevant to intrusion detection in a network?

II. How can the collected host information be applied to the alerts produced by signature based IDS such as Snort?

III. What are the benefits and drawbacks of introducing network awareness in an IDS, effectively making it a target based IDS?

1.1.2 Thesis outline

Chapter 2 provides background information on some key concepts pertaining to the work done in this project as well as information on the current state of the art in this topic.

Chapter 3 describes a number of different approaches to solving the problem statements provided above.

Chapter 4 describes how the prototype was implemented and tested

Chapter 5 displays the results from the testing stage as well as some information on the results

Chapter 6 contains a discussion of the approach, design, implementation and testing, as well as a discussion of possible future work

Chapter 7 summarizes the work done in this report and draws conclusions based on the answers found in each of the research questions. Some of the important contributions made in this project are also listed here.

Chapter 2

Technical background and literature

This chapter introduces the reader to some of the concepts and technologies used in this project.

2.1 A brief review of Intrusion Detection Systems

2.1.1 Signature based IDS

When a new security threat is discovered, a signature spesificly matching the patterns of the packets involved are produced to be used in a signature based IDS. Each packet passing through the IDS can then be compared with a set of known threats in order to determine if the traffic is malicious or legitimate. If the traffic matches one of the known signatures, an alert is generated and administrator is made aware of a possible security problem. An ordinary IDS is unable to investigate whether or not the threat is real, and so an administrator must manually investigate each alert to determine their validity.

A common tactic amongst automated attacks is to rapidly target random IP addresses in succession, attempting to exploit a set of frequently found vulnerabilities. Such attacks will, if matching a signature in the IDS be flagged as malicious traffic regardless of their success, leading to a high number of alerts corresponding to unsuccessful attack attempts. This inability to automatically determine the success, or even the possibility of a successful attack is the root cause of the poor true positive to false positive ratios in signature based IDS.

Consider a system consisting of publicly available Microsoft IIS web servers being targeted by an attacker. The attacker attempts to gain access to the machine using a buffer overflow attack designed to exploit a bug found in early Apache 2 versions in Ubuntu. Since neither the attacker nor the IDS has any information about the target machine, an alert is generated implying that the machine may have been compromised. In reality the attack could never have succeeded, but the system administrator will have to manually investigate the event.

If the IDS had known what systems the specific attack was designed for as well as the configuration of the target machine, the event could have been automatically ignored or an alert been generated with lower priority, allowing the administrator to work more efficiently.

To remedy these problems in such systems, it is common to spend a significant amount of time tailoring the IDS configuration to the system under protection. This could include removing signatures relating to Windows exploits in an environment consisting only of Linux servers, or perhaps specifying IP addresses and ports for certain IDS rules, for example defining which ports are used for HTTP servers. This approach is not without problems however, as even minor changes in the network topology or host configurations may lead to a miss-configured IDS where real attacks slip through. This means that the IDS configuration must change as the network changes in order to produce alerts according to expectations, and so this means that the system administrator must repeat the same tasks many times over.

2.1.2 Anomaly based IDS

A major issue with any signature based system stems from the fact that any new threat must be identified, patterns written in the form of a signature and finally distributed to the systems in order for detection to be possible. In contrast, anomaly based IDS attempt to determine what normal network traffic looks like, allowing the system to categorize the network traffic and detect deviations from what is perceived as normal. This means the system will be able to handle zero-day attacks or otherwise unidentified attacks. Anomaly based IDSs are not without their problems however, as they struggle greatly with correctly categorizing traffic in most real systems. Commonly anomaly based IDS have even greater problems with false positives than signature based systems.

2.1.3 Next generation IDS/Firewall

"Next generation" is a buzzword frequently used by network security vendors. It usually implies that the product offers entirely new capabilities in comparison to the existing solutions. Presently, from a network security point of view, these solutions usually include some way of providing context awareness by bringing together various solutions which have previously been separated, such as inventory or configuration databases, LDAP or Active Directory. The next generation IDS and firewall solutions work towards allowing more fine grained control of security policies and what parts of an application or service should be allowed by which users and at what locations. An example of this could be allowing the human resources department of a workplace access to the company's page on a social media site during work hours.

One of the recurring problems in many companies are related to users or employees circumventing company security policies either for convenience, efficiency or to be able to properly do their jobs. Cases like this is what the next generation systems work towards solving, as pointed out by Cisco's Senior Vice President[10].

Another challenge next generation products are working on is the increased use of encryption, especially on web services such as Gmail and Facebook where preventing session hijacking is imperative. In order for an IDS to inspect encrypted packets, they must first be decrypted, which requires cooperating with the local machine transferring the encrypted data. In 2011, Koch[11] investigated the requirements of a next generation IDS and the motivations for such a system.

2.1.4 Host and network IDS

Generally there are two different approaches to deployment of an IDS, each with their strong and weak points. By implementing an IDS on a host (HIDS) and allowing it access to the various files on the system, log files, an overview of running processes, their versions and network sockets as well as operating system details, better decisions can be made on whether a threat is real or not than what is normally possible in a network based IDS (NIDS). With direct access to the host, it is also possible to verify file integrity or detect when files have been added or removed, which again can help determine the success of an attack.

On the other hand, a NIDS has an advantage when it comes to detecting wide

spread attacks on several machines in the network. Consider a case where an attacker is scanning several neighboring machines in an attempt to identify weaknesses for exploitation. Another clear benefit of a NIDS is that enables protection of systems on the network regardless of the system administrator having access to the machines. The importance of this point becomes clear when considering the large number of networked mobile devices such as smartphones, laptops and tablets commonly used in institutions and workplaces.

2.1.5 False positives and false negatives

As earlier mentioned, Intrusion Detection Systems are prone to producing both false positives and false negatives. False positives, or type I errors occur when alerts are incorrectly generated where they should not have been, and while false negatives, or type II errors occur where malicious traffic does not raise an alert. These errors cannot be completely eliminated as long as we do not have correct signatures for all malicious traffic, while also having all corresponding host information available. Hence the best result we can hope for in such a system would be working towards reducing how frequently they occur.

Depending on the system and its context, the exact definition of what is considered a false positive may change. An alert may be considered a false positive if it was triggered by legitimate traffic containing the same signature as what we have defined as illegitimate traffic. Generalized signatures designed to match a variety of patterns are generally more prone to such errors than signatures matching a specific payload. Furthermore, an IDS may generate what could be considered an *irrelevant alert*. These irrelevant alerts are triggered by real illegitimate traffic, which due to their context are not interesting to the system administrator. This could for example be because the given traffic holds no significance when transmitted to a certain target. Such a scenario is mentioned briefly in subsection 2.1.1.

By adding another layer of signature based fingerprinting to the system, in order to gather host information for better decisions, we also introduce another error source. This can be seen to hold true for example in a scenario where a host is incorrectly assumed to be unaffected by an exploit, while it in reality is affected by it. This means that the host could be compromised without the IDS raising an alert, in this case causing a type II error which may not otherwise occur in an IDS system with manual alert verification.

2.2 A brief review of fingerprinting techniques

As the fingerprinting term suggests, it is concerned with identification of a target. Traditionally, scanning and fingerprinting tools have been useful to both network administrators and intruders looking for policy violations, weaknesses and security holes, as well as general network information. With the help of these tools one can learn a great deal about the hosts residing in the network. Examples being which services are listening to what ports, which ports are closed and which platform they are running on. In some cases further information about version numbers can also be extracted. All of this is valuable information whether one wishes to attack or defend a host.

Common for all fingerprinting techniques is that we require knowledge of what the traffic looks like for a given application or operating system in order for them to be identified. When done through a set of known signatures, the fingerprinting system becomes prone to false positives. If different systems produce similar datagrams, it may be difficult to distinguish them from another. An example of this can be problems distinguishing a Windows Vista machine from a machine running Windows 7 due to their many similarities.

2.2.1 Active fingerprinting

Active fingerprinting has long been used in well known network scanning tools such as Nmap[12], Nessusweb:nessus, THC Amap[13] and others. Common for the active fingerprinting techniques is that they generate network traffic. By sending packets to various ports and services running on a host, the response, or lack of response can often provide much information about the system. Active fingerprinting is however not without its flaws and weaknesses. It is not uncommon for experienced server operators to take measures to defend against such scans and greatly reducing the amount of information sent back to the scanning machine[14].

As a direct result of how the active scanning techniques work, they can cause the generation of significant network noise. Due to the time and resources it takes to scan a large number of ports and hosts, the scalability of active scanning techniques is also a concern. Furthermore, it would be reasonable to assume that the scanning systems' response time to changes in the network topology would also suffer.

2.2.2 Passive fingerprinting

Passive fingerprinting is a different approach to achieving the same goal. Tools such as PRADS[15], p0f[16] and Ettercap[17] have been developed to serve a need for network silence, requiring no additional network traffic to be generated. By reading package headers and contents from normal network traffic, these tools can often make precise guesses on what services the hosts on the network are providing. This solution operates very well in conjunction with an IDS machine where the network packages already pass through and are inspected by the machine.

2.3 Network Address Translation (NAT)

NAT is the process of modifying the information as defined in an Internet Protocol (IP) datagram. Commonly the IP address and/or port is modified, allowing a packet sent to one machine to be redirected to another host on the network. NAT has played an important role in the decline in available IPv4 addresses, and so is very widespread in many networks. NAT lets multiple machines behind a gateway utilize a single IP address by translating their private IP addresses to an external shared address as the packets pass through the gateway. Typically NAT is implemented by default in home and smaller business routers. Furthermore, NAT is used extensively in virtualized systems, where one might have hundreds or even thousands of machines sharing one or several IP addresses.

This may cause problems with any system attempting to define a host as an IP address. Rather than only using an IP address, it may be necessary to define a host as a combination of an IP address, port and protocol or any other property used for translating packets. Using the popular Linux firewall iptables[18], it is easy to define forwarding rules redirecting packets using these three characteristics. As an example, a package destined for the IP address 128.39.73.8 on port 1234, using the TCP protocol may be sent to an Ubuntu 10.10 host running an Apache2 web server. Meanwhile a similar packet headed for the same IP address and protocol, but using 1235 as port number might end up at a Windows 2008 machine running an IIS server.

2.4 Dynamic Host Configuration Protocol (DHCP)

One of the features of DHCP is that it lets unused IP addresses and other network information be assigned to new hosts on the network automatically. These addresses are often referred to as dynamic addresses, and is very helpful to seamlessly adding and removing mobile devices or hosts that do not provide a service to the network. This poses a challenge to systems seeking to track hosts though IP addresses, as the network may be constantly changing. An IP address assigned to a host that is fingerprinted as a Windows machine may for example refer to a Linux machine only moments later.

2.5 Previous work

The effort of bringing contextual or network awareness to the open source IDS solutions has been relatively unsuccessful, as can be seen from the fact that popular solutions such as Snort[19], Suricata and Bro[20] have yet to implement these features. Snort offers some means of describing host information in a network through their Host Attribute Table, it however relies on 3rd party software or manual inspection to provide the information. Snort is also very limited in how it makes use of the information. For the most part this solution provides a way to ease configuration, by automatically selecting which alerts will be used for which hosts, at the expense of maintaining an XML file with the host information. In order to alleviate some of the work required in maintaining such a file, some scripts have been written to translate the results of an Nmap or PRADS scan into a Host Attribute Table XML format[21, 22].

Snort's commercial parent, Sourcefire, has for many years been equipped with modules for providing real-time user and network awareness. Other competitors such as Palo Alto networks[23] and Cisco[24] are also heavily investing in providing contextual awareness for their solutions, however due to their commercial nature, few details are freely available on how these solutions specifically work.

Several different approaches have been attempted throughout the past 8 years to explore ways of reducing the number of false positives being generated in both Snort and general IDS implementations. In 2004, Kruegel et al[25]. investigated the possibilities of automatically validating alerts by actively scanning or accessing the host directly in real-time. The active scanning was done using Nessus[26], while the authenticated access was able to assure basic file and process integrity checks. Although not treated as a major concern, using both

of these methods in real-time as alerts are generated is slow in comparison to using already existing data from a local database.

Realizing that the majority of network traffic is web related, Karaarslan et al[27]. implemented a system for reducing false positive occurrences from web-attacks in 2006. This was done using a hybrid solution, utilizing both passive and an active fingerprinting techniques. It was concluded that actively scanning an entire network continuously would not be practical due to scalability issues and time constraints, and so their prototype was limited to a small set of common web ports. In other words such an implementation would not be feasible in a system where other protocols than HTTP are in use. Furthermore the passive fingerprinting portion was implemented using a custom Perl script, rather than publicly available tools. The passive fingerprinting was done by analyzing packages and looking for common web applications such as asp and php.

A third independent group explored the benefits of utilizing host data already present in a Configuration Management Database (CMDB) in 2008 [28]. A prototype based on Snort was tested on a known Internet Service Provider (ISP) network with real traffic, where all the hosts were described in detail in a CMDB. Because of this the prototype was shown to operate exceptionally well. The main concern in this system was the lack of detailed target definitions for some Snort rules, meaning the system was unable to make a decision.

In 2009, Sourour et al. brought up some important points concerning active scanning techniques[29]. It is noted that scanning can be intrusive or non-intrusive. While an intrusive scan would usually provide better information for the IDS, it is also more likely to interrupt the service, causing more problems than it solves. A solution is provided, implementing a solution closely tied to the Snort core, which relies on a host based Linux agent able to continuously extract all relevant host information. While the close ties to the Snort core allows this solution to operate in IPS mode as well as IDS mode, the paper does not discuss the assumption that a host based agent on every system to be protected is unreasonable.

A different attempt at reducing the number of false positives in a web was made by Mark Wierbosch in 2011[30]. Since the implementation discussed here is only concerned with unencrypted HTTP traffic, it is possible to use packet headers, HTTP status codes, user agent information and URLs to extract relevant information. This offers some benefits for decision making that would otherwise be unavailable in other IDS systems concerned with a variety of protocols. In contrast to other similar efforts, this implementation does not make use of operative system detection nor software and version num-

bers, however it is mentioned as a possible future improvement. Although the paper mentions using information reported by the web servers, it does not appear to recognize the fact that this information often can be spoofed with very little effort.

In addition to these attempts at reducing the number of false positives in IDS, additional research has been done on attempts to reduce the number of alerts reported as a whole. In 2007, Al-Mamory et al[31]. did a survey on the theoretical backgrounds of several methods of processing IDS alerts, focusing heavily on alert correlation. Amongst the discussed solutions were alert aggregation by grouping together similar alerts, finding the *root cause* of redundant alarms which are often caused by misconfiguration, and also a machine learning technique.

Spathoulas et al[32]. has an interesting approach where characteristics such as the fact that many attacks often probe several neighboring machines on a network to find vulnerable targets, and the fact that they usually have a recognizable alert distribution are taken advantage of. Rather than filtering false positives out, true positives are "filtered in" using these characteristics. Since the approach is heavily based on observed alert distributions and expected attack characteristics, it is prone to erroneous decisions where real attacks are removed when the assumptions are not met.

2.6 Where does this thesis fit in?

By providing the necessary host and rule information for making better decisions to the IDS, the hope is to drastically improve the false positive to true positive ratios. In turn, it is assumed that this will help improving both the response times and work gratification for the system administrator tasked with investigating the alerts. Furthermore, allowing the IDS to automatically make better decisions based on basic rule and host information will improve the out-of-the-box state of an IDS, reducing the amount of time spent on both initial configuration and maintaining the IDS. Most system administrators recognize the importance of automating and eliminating tedious day-to-day tasks. As such, both the false positive reduction and less dependency on configuration should be well received by the system administration community.

The proposed implementation will bring together existing free and open source solutions in a useful and reproduceable manner. Additionally, the results will demonstrate whether or not such an implementation is helpful and to what extent. More research in this area of intrusion detection systems will also hope-

fully help bring more attention to its benefits, and thereby giving incentive for IDS developers do adopt such a solution.

Another side effect of bringing context and network awareness to the current open source IDS systems, is that it will bring them closer to what is commonly perceived as "next generation" IDS. Once network awareness is available it would not require as much effort to further expand the solution to for example incorporate more fine grained controls.

Chapter 3

Approach

In this chapter the various approaches are discussed in order to find the most suitable way to solve the research problems as well as providing a fixed plan for the project.

3.1 Introduction

There are many possible approaches to providing answers to the problem statements discussed in chapter one. Furthermore it is necessary to identify the functionality required by the system and how it can be realized. It is also important that the effect of the final results can be evaluated formally in order to determine the success of the study. The approach seeks to bring to light any problems with the different approaches, so that they can be avoided at an early stage of the project.

3.2 Preliminary decisions

One of the first decisions to be made is whether to create a new network IDS with the proposed network awareness feature, or provide an addition to an already existing tool. Given the time constraints for this project, it seems unlikely that implementing a new unsophisticated IDS for testing purposes would be beneficial. Time spent on designing, implementing and testing such a system could be better spent focusing on the new feature. Snort[19] and Bro[20] are both mature open source projects in active development, and are both attractive options for use in this project. Having previous experience with

the Snort IDS and general familiarity with this software means that more time can be spent on implementing the prototype. The great popularity of Snort and the excellent official as well as third party documentation is bound to be helpful should problems arise at a later stage of the project.

In 2011, Jonas Taftø Rødfoss[33] explored three popular IDSs. After taking into consideration the problems he experienced using Bro in a similar environment, Snort was selected as the primary choice in this project. Both Snort and Bro are single threaded applications, though high performance is not considered a high priority in this proof of concept implementation.

3.2.1 Data collection

Collecting host information from each networked host under our IDS's protection is an integral part of the proposed network awareness feature. Chapter 2 mentions two different ways of collecting host information by reading datagrams and comparing them to a set of known patterns, commonly referred to as signatures.

Active fingerprinting

The active approach, relying on sending a stimuli to the target, hoping to provoke a specific response is perhaps the most effective solution. Being able to control the stimuli being sent to the target is very beneficial, because it often allows us to trigger a certain expected response. This in contrast to relying on normal network traffic to generate the necessary data when passively fingerprinting.

Some drawbacks of active scanning were also mentioned in chapter 2. In order to ensure that all target information is available when determining the validity of an alert, it is necessary to either continuously scan all hosts, on all ports, on various protocols, or actively scan the target during the validation process when one knows where and what to scan for. Clearly the continuous approach would allow us to already have the data at hand when needed, allowing for quick decisions when data has already been collected. The data collection process itself however would grow rapidly as the number of hosts increases, meaning it would not be scalable. Actively scanning as a part of the validation process on the other hand would be expected to slow it down significantly in comparison to for example reading host information from RAM.

Passive fingerprinting

The other approach mentioned, passive fingerprinting, does not necessarily have either of these problems. The packages already passing through and being read by the IDS host can be used for fingerprinting, allowing for excellent scalability. This of course means that no host will be visible by the system until an appropriate package has triggered a signature in the fingerprinting system. As mentioned in the background chapter, another benefit of the passive approach is that it is completely silent on the network, allowing for a very stealthy and "clean" system.

Host based agent

There are however other ways to collect host information besides the above mentioned fingerprinting methods. One of these methods relies on authenticated access to each host we would like to extract host information from, and is perhaps the most obvious as well. By installing a script, hereby referred to as an "agent" on each target machine we are able to extract extensive and reliable information. On most Linux machines for example this could be done by creating a script to extract detailed information about the operating system, running applications listening for incoming traffic, as well as sockets, versions and even patches.

As mentioned in the background chapter however, it would be unreasonable in many networks to assume that we have full access to every host on the network. Scalability is also a concern with such an implementation, as the agent would have to be installed on each target. This burden could of course be managed efficiently by utilizing a configuration management system such as the widely used Cfengine[34] or Puppet[35]. Another possibility could be to install the agent though a Pre-boot Execution Environment (PXE)[36], which allows several machines to be set up with identical initial configurations automatically over a network.

Furthermore, with an agent residing on the host it collects information from, the system may be exposed to employees or other users tempering with the system. In essence, this means that users, depending on their access rights may be able to for example change the files the system reads data from, or change the output returned by certain commands. A rogue employee may therefore be able to control which alerts are being displayed to the system administrator for that host by making the IDS think a given alert would not be applicable to that target. By manipulating files the agent relies on so that the agent reports a

different operating system is present, alerts against this host may be incorrectly presumed to be irrelevant by the system.

While this might seem like a minor concern at first, respectable sources such as Verizon[37] and Computer Security Institute (CSI)[38] both recognize the danger of insider threats, although to different extents. The reported figures differ greatly, due to different data sources and difference in what is perceived as an insider attack.

Configuration Management Database

A final approach, utilizing a Configuration Management Database would allow the system to verify alerts triggered by traffic to any host defined in the database. This solution however shares some concerns with the agent based solution, as the system will know nothing about hosts not managed by the configuration management system. Furthermore, it is not uncommon that machines are only partially managed by the configuration management system, meaning that the system may only know about a small set of services running. As briefly mentioned in the related work section, this solution was investigated by Pimenidis et al. in [28] with excellent results, largely due to having a good Configuration Management Database available.

3.2.2 PRADS - Passive Real-time Asset Detection System

The passive fingerprinting method was chosen after carefully considering the pros and cons of the various data collection methods, as discussed above. The scalability and stealthiness, and of course the fact that very little public research appears to have been done in this specific area. It is hoped that by using the passive approach, the proof of concept in this project will provide a more realistic implementation than the previous attempts at network awareness in IDS mentioned in the previous work section.

There are several publicly available and open source passive fingerprinting tools. PADS (Passive Asset Detection System), not to be confused with PRADS, is an old tool, discontinued in 2005 and mainly focused on detecting active services on the network, rather than determining which operating system is running. On the other hand, p0fv2 is popular signature based operating system fingerprinting tool, last updated in 2006. Inspired by these two, PRADS was implemented and is still in active development. This tool is able to use

signatures from p0fv2 and PRADS directly, and is able to detect both services and operating systems.

Very recently, p0fv3[16] was announced as a complete rewrite of the previous version. Since this tool is still very immature and also incompatible with previous signatures, it was not considered a viable option for this project. In the future however when a sufficient amount of signatures have been provided, this tool could be a very attractive option.

SinFP[39] is a passive fingerprinting tool, which explicitly states that it recognizes the limitations in other similar tools when dealing with Network Address Translation. Like p0fv2/v3 however, this tool is only able to detect operating systems, and would therefore have to run in addition to either PADS or PRADS to get a full overview.

For these reasons, PRADS was chosen to be the primary data collection tool in this system.

3.3 Lab environment

In order to test the proof of concept and verify its success (or failure), it will be necessary to run Snort on a network in order to generate some alerts. It is assumed that having a large number of hosts and alerts, with both true and false positives amongst them will make it easier to determine the effect of the proposed feature. A lab environment is provided by HiOA with several hundred machines with public IP addresses as depicted in figure 3.1. A large portion of the servers reside in virtualized environments used by students in various courses. For the most part these virtual networks consists of one Ubuntu host with a public IP address, serving as a gateway for one Windows XP or Windows 7 host, and possibly additional Ubuntu or Debian servers. This means that it will be critical that the prototype is able to handle Network Address Translation in a meaningful way.

By configuring a "span port" on the main switch, all packets passing through its outgoing port will be mirrored and sent to the server running Snort and the fingerprinting software. This way, this host will be able to sniff and analyze all traffic passing through this choke-point by configuring the network card to run in promiscuous mode. This second network card does not require an IP address in order to listen to the traffic being sent to it by the switch. By severing the other connection from the switch to the network card labeled `eth0`, the machine would be extremely difficult to compromise for an attacker, but

require the system administrator to have physical access. These are important points from a security point of view.

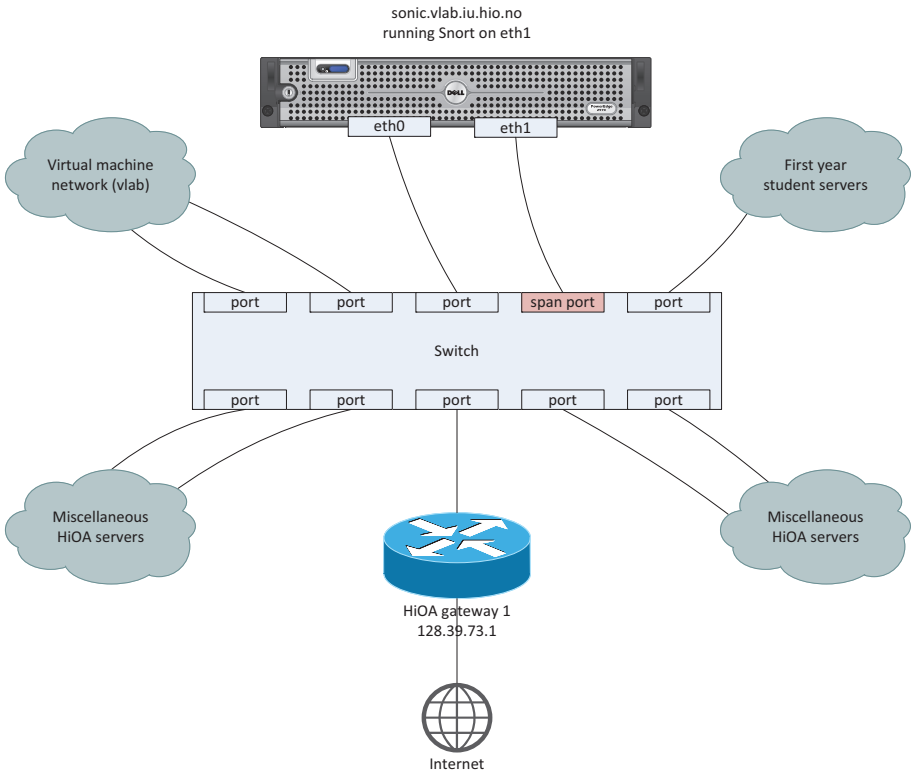

Figure 3.1: The lab environment used for testing the prototype.

3.4 Prototype

The first step necessary in order to evaluate the effectiveness of adding "network awareness" to an intrusion detection system is to develop a prototype with such a feature. As previously mentioned Snort was selected to be the IDS in which the new feature will be added. Snort is able to operate in either in-line mode, otherwise known as intrusion prevention mode, or as an intrusion detection system. For an IDS to run in in-line mode, it is imperative that the system is as efficient as possible when deciding whether to allow a packet to pass through or if it should be intercepted. As can be seen from the lab environment shown in figure 3.1, running the prototype on an IPS system will not be possible, as all traffic passing through the switch is simply copied and sent to the second network card on the machine labeled *sonic.vlab.iu.hio.no*. Furthermore, it would not be desirable to test such a system in a production environment. For these reasons IPS mode will not be considered in this project and so the performance of the alert validation process can be given a lower

priority.

Commonly Snort is configured to log all alerts to a binary file, rather than inserting the data directly in a database. This is done to improve the Snort performance, reducing the chance of packets being dropped by the system. Barnyard2 is a program which compliments Snort by running at certain intervals, parsing the binary data and inserting it into a database such as MySQL.

There are a few possible ways to ensure that new alerts are validated against the host information database(s). Firstly, we could modify Snorts source code directly so that the validation is done as a part of Snorts alert generation process. Barnyard2 could then be modified to handle the new data accordingly and insert it into the database along with the default information. This however would be a difficult task, as the Snort and Barnyard2 source is written in C and also rather complicated.

An easier implementation would be having an external program which is able to run at certain intervals, or be triggered at the same time Barnyard2 is executed. It should be noted however that some systems use SMS or Email warnings when certain incidents happen, and such warnings should not be issued until after the validation process is done. In other systems it is common to use a web interface to view IDS alerts. In such a system the validation process might not be as time sensitive, as new alerts could simply be displayed as not yet validated for a few seconds.

The planned system design is described in figure 3.2. The Perl script `validator.pl` provides the new functionality by comparing information about a given alert to the host information of the target. The script can be written in a modular way, so that it can utilize various host information sources. Information about the various rules and the systems they are applicable to can be gathered from various websites such as the Common Vulnerability Database (CVD)[40], the Snort home page[19] or Bugtraq[41].

Perl is a widely used high-level interpreted programming language, favored by many system administrators due to the ease of implementing advanced functionality with minimal work. Perl is also well known for having strong text processing aptitudes, which will be important in this project as it will rely heavily on processing and comparing strings. Through the Comprehensive Perl Archive Network (CPAN)[42], Perl modules are provided for common tasks such as interacting with databases or establishing network connections. These modules can be easily installed from the command line on demand and used in Perl programs.

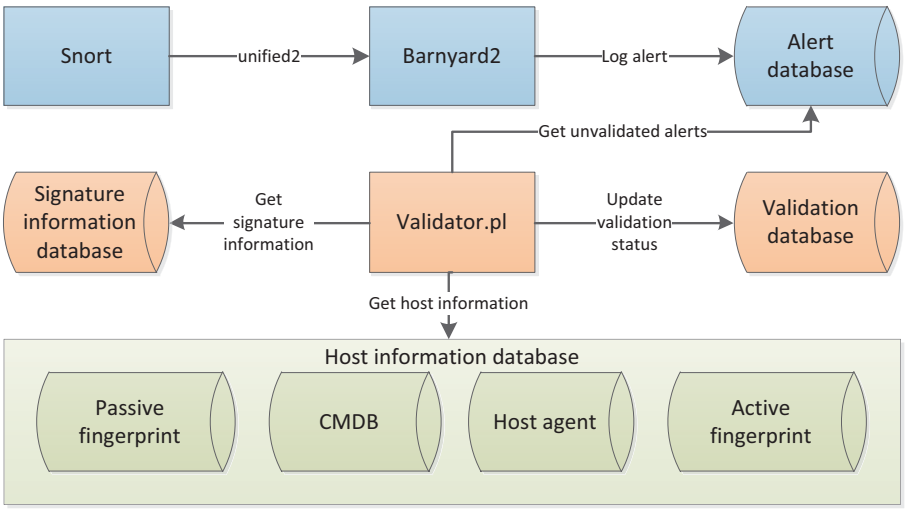

Figure 3.2: Proposed system design for the prototype.

3.5 Testing the prototype

After carefully designing and implementing a prototype with the desired functionality, it is crucial that we test it in the lab environment and that the results will be able to provide answers to the problem statements. At first, the command line tool tcpdump will be used to sniff, collect and log all the packages traversing the main switch to a log file which can be read and analyzed at a later point. Once the binary file has grown sufficiently large, it can be interpreted and analyzed by Snort and PRADS in order to produce Snort alerts and PRADS host information. The Perl script `validator.pl` can then be executed, and proceed to estimate the relevance of each alert.

We may then compare the default Snort alerts with the validated alerts. The success of the proof of concept will be determined based on how well the new information added to each alert allows the system administrator to separate the relevant alerts from the irrelevant alerts. The desired outcome will be that the alerts are categorized according to how one might expect to categorize them using manual inspection.

Chapter 4

Methodology and implementation

This chapter describes what was done during the implementation stage, as well as how it was done. Choices made during implementation are also documented here along with their reasons.

4.1 Lab setup

With the purpose of improving the reproducibility of the system and the experiments performed in this project, some of the details regarding the software setup and their configurations are described in the following sections. Default configurations can be assumed to be omitted from this report.

The host referred to as `sonic.vlab.iu.hio.no` in figure 3.1 had properties as described in table 4.1.

Item	Property
FQDN	sonic.vlab.iu.hio.no
Machine model	Dell PowerEdge 2850 Server
Processor	64-bit Intel Xeon CPU 2.80GHz (Dual core)
RAM	4x 512MB + 2x 1024MB DDR2 at 400 MHz
NIC	2x Intel 82541GI Controller with 1 Gbit/s capacity
Operating system	Ubuntu 11.10 Oneiric Ocelot x86_64
Linux kernel	3.0.0-12-server

Table 4.1: Server properties

4.1.1 Snort

Snort version 2.9.2.1 and its prerequisite data acquisition API `DAQ` version 0.6.2 were installed from source with the following options.

```
./configure --prefix=/usr/local/snort --enable-perfprofiling --
    enable-decoder-preprocessor-rules --enable-ppm --enable-
    sourcefire
```

Code 4.1: Snort compile options

In previous versions, Snort required tweaking of several compile options to operate optimally, however in recent versions these settings have been made default. Snort was then configured to log its output to a `unified2` format binary file for further processing by Barnyard2. Barnyard2 was then configured to read the `unified2` format files and insert the data into a MySQL database. An official Snort rule set from april 12, 2012 was used, with the exception of the precompiled rules which were dismissed.

The default rule sets `policy.rules` and `file-identify.rules` provided several hundred signatures referring to possible policy violations such as suspicious DNS queries, BitTorrent downloads and cloud storage synchronization traffic. Another problematic rule set was the `icmp-info.rules` file, which contained almost 100 different ICMP traffic signatures, many of which were general ping requests and replies. These alert files were disabled since they were not interesting for testing the alert validation feature, since a simple ICMP echo request is valid against any target able to handle the ICMP protocol. These alerts served little purpose besides generating several thousand alerts per hour.

Due to the high number of connections Snort had to track when analyzing the network traffic, some tweaking had to be done to allocate more memory to

the the Stream5 preprocessor. The modified line is shown in code listing 4.2 with the new memcap value, which was set to 16 times higher than default. After these modifications no error messages were shown and packages were no longer being dropped when Snort could not keep up.

```
preprocessor stream5_global: track_tcp yes, memcap 134217728,
    track_udp yes, track_icmp no, max_tcp 262144, max_udp 131072,
     max_active_responses 2, min_response_seconds 5
```

Code 4.2: Modifications to the snort.conf file

4.2 Prototype implementation

A prototype was implemented in Perl, which analyzes each alert produced by Snort and compares them to data gathered by PRADS as well as information about the signature which triggered the alert. The final implementation of this prototype can be seen in its entirety in appendix A. The early system overview depicted in figure 3.2 shows that the prototype has access to a host information database (in green). This database could hold information from several different sources, with varying degrees of correctness. The way the final prototype was implemented, a single host data source from the host information database is used at a time. As mentioned in the approach chapter, PRADS was chosen as the host data collection tool, however any source could be used without program modifications as long as the database follows the same format as PRADS.

Initially, the intent was to design the prototype in a way that allowed it to automatically gather all the information necessary to make appropriate decisions using already existing datasources. This however proved very difficult or even impossible given the time frame for the project. The problems with automatically gathering signature and host information as well as the solutions to these problems are described in subsection 4.2.1 and subsection 4.2.2.

4.2.1 Gathering signature information

Most Snort rules contain references to online sources where information about the given vulnerability is described. Frequently occurring references include Microsoft technet security bulletins[43], Bugtraq[41], Common Vulnerability Database (CVD)[40] and Snort's home page [19]. There is no standard for how these references should provide the information, and so it varies greatly what

information is displayed and how it is structured. Of the previously mentioned reference sites, CVD had a clear advantage in that it offered a downloadable database in several formats including XML and comma separated values (CSV). This was attractive because it meant the script would not rely on Internet access for lookups.

The prototype would then have the goal of providing a best-effort service given the host, signature and alert data which was available to the program. What this meant was that the prototype could not guarantee that every alert which has corresponding host and signature information would be validated based on this data with as good results as when a human was doing the same job. Given the complexity of the often arbitrary data the prototype would have to read and understand, it would utilize this information with as much precision as a "dumb" program would permit.

To better understand why the prototype would inherit this limitation, consider a Snort alert with a corresponding CVD vulnerability description as seen in code 4.3. A human reader might be able to deduce that the version numbers mentioned refer to Adobe Flash Player, rather than a version number for the Linux Kernel, despite the fact that the numbers are directly following "Linux" in the description. This reasoning would be possible if one knew that the most recent kernel version as of time of writing was 3.0.31, much lower than the number seen in the description.

```
Unspecified vulnerability in Adobe Flash Player for Linux
10.0.12.36, and 9.0.151.0 and earlier, allows remote
attackers to execute arbitrary code via a crafted SWF file.
```

Code 4.3: CVD description from the ID CVE-2008-5499 (SID 15869)

Similarly, the Snort website produced the information seen in code 4.4 about the same rule. Note also that the section headers such as "Affected Systems" used on the Snort page were missing from several rules.

```
Summary
  This event is generated when an attempt is made to exploit a
  known vulnerability in Flash Player for Linux.
Impact
  Denial of Service. Information disclosure. Loss of integrity.
  Complete admin access.
Detailed Information
  Unspecified vulnerability in Adobe Flash Player for Linux
  10.0.12.36, and 9.0.151.0 and earlier, allows remote attackers
  to execute arbitrary code via a crafted SWF file.
Affected Systems
  Adobe Flash Player for Linux 9.0.115.0
```

```
Ease Of Attack
  Simple. Exploits exist.
False Positives
  None known.
False Negatives
  None known.
Corrective Action
  Upgrade to the latest non-affected version of the software.
  Apply the appropriate vendor supplied patches.
```

Code 4.4: Snort.org vulnerability description from the SID 15869

It was clear that the data in the signature references was intended to be read by humans rather than a program. Traditionally IDS tools have used web interfaces such as BASE[44] or other ways to directly query the alert database or log file. Since these alerts are then manually investigated by a human reader, it hasn't been a high enough priority to provide an API or strict format on the external alert information.

To solve these problems, a new table (vulnerabilityinfo) was created in the Snort database providing strictly formatted vulnerability information for each alert seen by the system. The information in this table was gathered by manually inspecting the references for each unique rule triggered by Snort during testing. This solution was possible since only a few rules were triggered by Snort, and would not be a practical implementation for longer Snort runs in a real world scenario.

The format for target vulnerabilities was defined closely matching the structure already present for some fingerprints used for operating systems in PRADS. Two examples of this format can be seen in code 4.5. One or several operating systems with zero or more corresponding versions could be defined this way. The same formatting was used for service definitions.

```
operating system 1:firstversion/secondversion!operating system 2

service 1:version 1!service 2: version 1/version 2
```

Code 4.5: Definition format used in vulnerabilityinfo

One minor limitation imposed by this format however, was that services could not be defined per operating system type. This would be problematic if the same vulnerability was present in the TFTP client FileZilla on Windows and the TFTP client VSFTPD on Linux for example. Such cases are very rare however and no such vulnerabilities were encountered during testing, and so this limitation had no practical impact for this report.

Generally when designing databases it is advisable that each value is defined atomically. This means that the data cannot be further split into new values. The information held in the vulnerabilityinfo table can clearly be represented separately in some cases. It would be possible to have one operating system on each row, rather than one row with multiple operating systems. There would however be little to gain from doing this in this particular system, due to the large number of additional reference tables required to represent the information in the same way. This is because of the interdependencies on the operating system, service and version fields for the various vulnerability definitions.

Having all the values in two separate columns with a consistent format for separating the data using meta characters makes it very easy to extract the information using *regular expressions* (regex) in a Perl script.

4.2.2 Gathering host information

PRADS writes its information to a log file with a strictly enforced comma separated format as seen in code 4.6. The meta tag contains information from the fingerprint or signature which PRADS detected and which caused the entry to be written. The service field indicates the fingerprint type. These can be `SYN`, `SYNACK`, `RST` or `FIN` depending on the flags set in the TCP datagram, or `SERVER` and `CLIENT` depending on the type of service detected.

```
ip address,virtual lan tag,port number,protocol number,service,[
    meta],distance,time discovered
```

Code 4.6: PRADS CSV log format

Code listing 4.7 shows three valid fingerprint entries as seen in the PRADS log file. Enclosed in the meta character square brackets `[` and `]` is the information we wish to extract. Some of the characteristics of the data within the brackets are as follows: the number of separating colons `:` varies slightly, and most importantly the version string for an operating system varies greatly. In the second entry, commas `,` are used as separating characters between the versions, while the third entry uses a slash `/`.

```
128.39.73.*,0,443,6,SYNACK,[S4:64:1:52:M1460,N,N,S,N,W7:ZA:Linux
:2.6 (newer, 7):link:ethernet/modem],0,1328570603

128.39.73.*,0,55076,6,SYN,[8192:64:1:48:M*,N,N,S:.:Windows:2000
SP2+, XP SP1+ (seldom 98, Low TTL1):link:ethernet/modem:uptime:
1002hrs],0,1328570784
```

```
80.202.171.*,0,61393,6,SYN,[8192:119:1:52:M1452,N,W2,N,N,S:.:
Windows:XP/2000 (RFC1323+, w+, tstamp-):link:pppoe (DSL)]
,9,1328570614
```

Code 4.7: PRADS log example (anonymized IPs)

In order to extract information relevant for making a decision in the validating program, it is necessary to know where in the fingerprint the data resides, or what the information we are looking for looks like. That way, we can extract the substrings we are interested in, using regular expressions (regex). To achieve this, all fingerprints for both services and operating systems were modified slightly. A few examples of default PRADS signatures for operating system detection are shown in code 4.8.

```
1  S4:64:1:52:M*,N,N,S,N,W0:ZA:Linux:2.6 or 2.4 w/o timestamps
2  5712:64:1:40:M*,S,T,N,W4:ZAT:Linux:2.6 (newer, 4) IPv6
3  S12:126:1:52:M*,N,W0,N,N,S:A:Windows:2000 (SP1+ Userapp2?)(UC)
4  8192:128:1:56:M*,S,T:A:Windows:2000 SP2+, XP SP1+ (seldom 98)
5  65535:64:1:60:M1368,N,W3,S,T:AT:FreeBSD:FreeBSD 20061110
6  S6:64:1:60:M1460,N,W0,N,N,T:AT:BSD/OS:4.0.x
7  33304:64:1:60:M*,N,W0,N,N,T:AT:MacOS:X 10.2.6
8  16384:128:0:60:M1460,N,W0,N,N,T0:A:Akamai:??? (2)
9  16384:128:0:48:M1460,N,N,S:A:unknown:something
10 252:128:1:40:.:AFN:Windows:Windows 7/2008 R2
11 0:255:0:40:.:.:Linux:2.0/2.2 or IOS 12.x (dropped)
```

Code 4.8: Default PRADS OS signatures and fingerprints

The same 11 lines are shown in code 4.9 where the problematic aspects of each line has been corrected. While modifying the lines, curly brackets `{` and `}` were added surrounding the relevant information, for easier extraction by the validating program.

```
1  S4:64:1:52:M*,N,N,S,N,W0:ZA:{Linux:2.6/2.4}
2  5712:64:1:40:M*,S,T,N,W4:ZAT:{Linux:2.6}
3  S12:126:1:52:M*,N,W0,N,N,S:A:{Windows:2000 SP1+}
4  8192:128:1:56:M*,S,T:A:{Windows:2000 SP2+/XP SP1+/98}
5  65535:64:1:60:M1368,N,W3,S,T:AT:{FreeBSD:20061110}
6  S6:64:1:60:M1460,N,W0,N,N,T:AT:{BSD:4.0}
7  33304:64:1:60:M*,N,W0,N,N,T:AT:{MacOS X:10.2.6}
8  16384:128:0:60:M1460,N,W0,N,N,T0:A:{Akamai:0}
9  (Removed entirely, no useful information at all)
10 252:128:1:40:.:AFN:{Windows:7/2008R2}
11 0:255:0:40:.:.:{Linux:2.0/2.2!IOS:12}
```

Code 4.9: Modified PRADS OS signatures and fingerprints

Words clearly intended for human reading such as “or”, “seldom”, “something” as well as question marks were removed and replaced by appropriate meta characters separating the data fields. A similar format to what PRADS had already been following for some entries was used. To separate multiple operating systems an exclamation mark `!` was used, since PRADS did not have a meaningful way to separate multiple possible operating systems already. Furthermore, a colon `:` was used to separate an operating system from it's version, as was already being done by PRADS. To separate the several possible operating system versions a slash `/` was used.

The version number 4.0.x was translated to 4.0, and ??? was translated to 0, meaning we don't have any version information. Knowing that there might be an unknown subversion of a service holds no meaning to the prototype, and might even cause problems when the prototype attempts to compare this version to another. Other irrelevant information such as a packet being dropped or a datagram using IPv6 was simply removed.

Similar modifications were done in the fingerprint files for services for them to be consistent and also ease data extraction. Code listing 4.10 shows the original fingerprints followed by a regular expression signature, while code listing 4.11 shows the modified fingerprint.

```
1 ssh,v/OpenSSH/$2/Protocol $1/,SSH-([.\d]+)-OpenSSH[_-](\S+)
2 http,v/Apache/$1//,Server: Apache\/([\S]+)[\r\n]
3 http,v/Apache/$1/$2/,Server: Apache\/([\S]+)[\s]+\((.*)\)
4 http,v/Microsoft-IIS/$1//,Server: Microsoft-IIS\/([\S]+)[\r\n]
5 http,v/NetCache//$1/,Server: NetCache (\(.*\))
6 ftp,v/Microsoft FTP Server/$1//,Microsoft FTP Service \(Version
    ([\S]+)\).
7 ftp,v/Microsoft FTP Server///,220 Microsoft FTP Service
```

Code 4.10: Default PRADS service signatures and fingerprints

```
1 ssh,v/{OpenSSH/:$2}/Protocol $1/,SSH-([.\d]+)-OpenSSH[_-](\S+)
2 http,v/{Apache/:$1}//,Server: Apache\/([\S]+)[\r\n]
3 http,v/{Apache/:$1}/$2/,Server: Apache\/([\S]+)[\s]+\((.*)\)
4 http,v/{Microsoft-IIS/:$1}//,Server: Microsoft-IIS\/([\S]+)[\r\n]
5 http,v/{NetCache}//$1/,Server: NetCache (\(.*\))
6 ftp,v/{Microsoft FTP Server/:$1}//,Microsoft FTP Service \(
    Version ([\S]+)\).
7 ftp,v/{Microsoft FTP Server}///,220 Microsoft FTP Service
```

Code 4.11: Modified PRADS service signatures and fingerprints

It should also be noted that PRADS removes the `/` meta character from the fingerprints when they are reported in the log file, which might cause con-

fusion when attempting to extract a version number following the service in some cases. Adding the enclosing meta characters `{` and `}` corrected this problem.

A Perl script (prads2db.pl) included with PRADS was used to insert the log file data into a MySQL database. After all this was done, the PRADS data was ready to be used in the prototype through MySQL.

4.2.3 Version number comparison

Version numbers and version names are used to provide unique identifiers of a software's development state and is common to virtually all software. There are many conventions and standards amongst different organizations and communities, noteworthy examples being the GNU/Linux community and Microsoft developers. A very common way to present version numbers is through a sequence based identifier such as `9.0.7.1`. Each sequential number has special meanings, which may vary entirely dependent on the software author. This version number could sometimes be interpreted as follows:

- The first number (9), usually refers to a major version of the software, often incremented after major changes have been made to the software.
- The second number (0) then refers to a minor version within this major version.
- The third number (7) commonly refers to a specific build number within the minor version, and is often more meaningful to a developer than end users.
- The fourth number (1) could then refer to a revision of this build.

Table 4.2 shows a list of some common version numbering schemes used by software developers.

Clearly there are very many different formats which might occur in either PRADS or the vulnerabilityinfo database, and in turn must be understood by the prototype. A large portion of these formats are common numerical formats used on Linux systems and in general applications, and the named versions generally used for Microsoft Windows. The nature of the sequential numeric representations makes it easy to compare all of these with little effort, assuming date formats are written in descending order. Given that such

Formatting	Description
`9.0.7.1`	A simple and commonly used formatting
`9.0.7b2`	The letter `b` and number 2 could mean that this is a beta with some bug fixes
`9.0.7rc1`	`rc1` usually means release candidate 1
`9.0.7r2`	`r2` often means that this is a commercial release with some improvements
`9.0_7.1`	Another way to separate the version numbers
`Vista`	Naming used as a version identifier, refers to Windows Vista. Also has a version and build identified as 6.0.6002, though this number is shared with Windows Server 2008
`Vista SP2`	Naming used as a version identifier, with a second portion defining a minor version
`Office 2010`	Naming and year used as an version identifier
`2.5.10/build 69`	A version number providing a clear meaning of the last digits
`2012.05.13`	A date used as a version number

Table 4.2: Examples of various software versioning schemes

comparisons are basic methods frequently used in several applications, CPAN provided several modules with sufficient functionality.

The Perl module named `Sort::Versions` was installed using the command `cpan -i Sort::Versions.pm`. This module allows comparison of version numbers delimited by periods `.` and hyphens `-`, including versions containing alphabetical characters. Other characters and special signs may however have unpredictable results as they are compared used ASCII ordering, meaning that the binary representation of ASCII code characters are used.

Generally when comparing a version found in a PRADS fingerprint with a version found in the vulnerabilityinfo table, the versions will be on the same format. Due to this, the `Sort::Versions` module becomes even more powerful, and will allow comparison of version names where the same character sequence or string occurs in both versions. This is the case in Windows service packs, which can be represented as `SP1`, `SP2` and so forth.

This method will however not be able to handle version names where the character sequence holds special meanings, such as in the Windows versions “XP” and “Vista”. To handle these cases, one possibility, and also the method implemented was to define a static indexed array in Perl, storing every known Windows version name in ascending order. That way we would have a lookup

table and could simply compare the array indexes. The minimal example show in code 4.12 would return 6 when executed with `XP`, while `Vista` would return `10`. That way we know that Vista is greater than XP.

```
1 my @windows = qw{95 NT4.0 98 98SE 2000 Me XP XP-64 2003 XP-Pro
      Vista 2008 7};
2 my %index;
3 @index{@windows} = (0..$#windows);
4
5 my $find = <STDIN>;
6 chomp $find;
7 my $index = $index{$find};
8 print "$find is windows version $index\n";
```

Code 4.12: Perl indexed array example

Despite Windows operating systems being marketed and associated with various version names, each release still retains an internal version number referring to the kernel version and build on release. Windows XP for example has the version number `5.1.2600`, while Windows 7 has the version number `6.1.7601`. Since the indexed array implementation for version comparison requires that the program is updated on new releases, an alternative solution was considered.

Rather than specifying operating system version names in the fingerprint files and vulnerabilityinfo table, these names could be changed to hold the numeric version discussed above. That way whole strings such as `6.1.7601 SP1` could be compared with `6.1.7601 SP2` or `6.0.6002 SP1` and still report correct results using the earlier mentioned Perl module.

The reason this method was not implemented is because it caused problems in a few special cases. Some Windows operating systems such as Windows Server 2008 R2 and Windows 7 share the same kernel version. Generally this is not a concern, due to the major similarities between these versions, but it still may occur that a vulnerability is only found in one the two systems. It would also require yet another modification to the PRADS fingerprint files.

4.2.4 Databases

Three new tables storing necessary information to the prototype were added to the MySQL database. Their relationships with the already present default Snort tables are described in figure 4.1. The figure is simplified in that it only shows the fields and tables relevant to the new features provided by the proto-

type. At the core of the system is the event table, which together with the other Snort tables define unique alerts. Snort tables omitted from the figure include references to online signature information, datagram payloads and header information for ICMP packets. The reasons for not handling ICMP alerts are described in subsection 4.2.5.

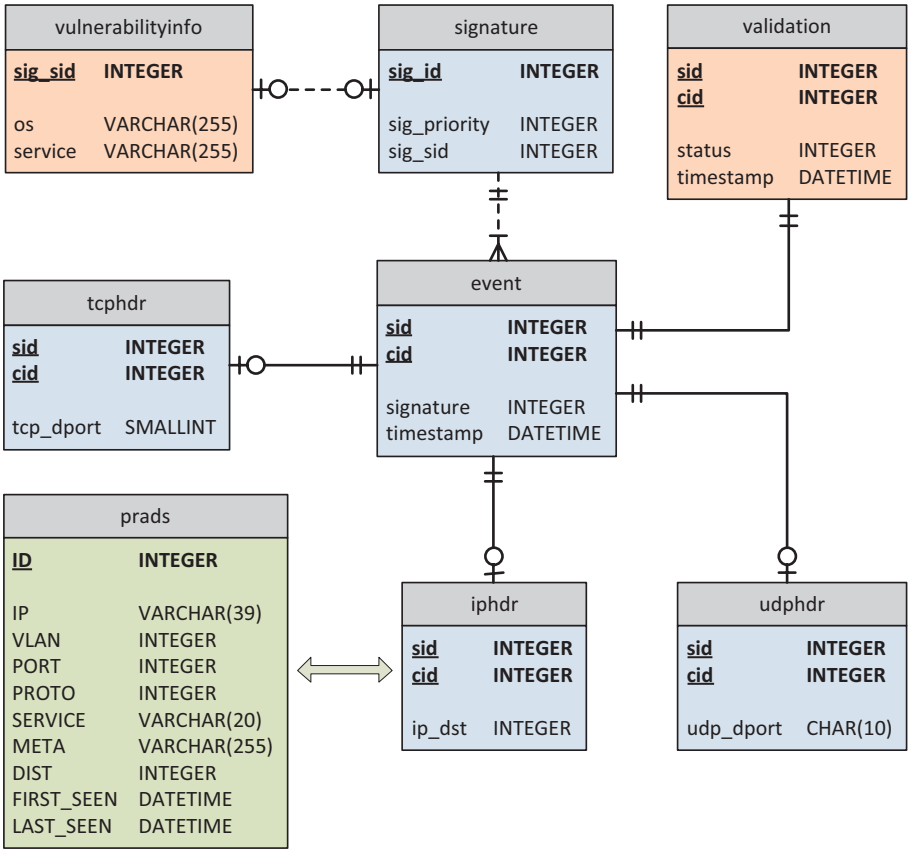

Figure 4.1: A simplified EER diagram for the system's databases and tables.

The `prads` table is linked to a Snort event's IP address in `event` and a port number found in either a TCP (`tcphdr`) or UDP (`udphdr`) header, depending on the protocol used. Each unique signature also has a corresponding entry in the `vulnerabilityinfo` table.

The prototype, when executed, inserts a new entry into the `validation` table for each unique Snort alert. The contents of this validation table are the end results of the prototype, and defines whether an alert could be ignored or if it corresponds to a real threat.

4.2.5 NAT considerations

Network Address Translation has been an extremely important aspect of this project, imposing limitations in the amount of host information available in PRADS. As briefly described in section 2.3, NAT allows several machines to share a single IP address by rewriting package headers at a gateway. Figure 4.2 illustrates how a NAT-ed part of the network might affect the alert validation system. To correctly identify a host in such environments, the prototype uses an IP address, port number and the protocol type to uniquely define a host.

With the network in figure 4.2 as an example, the prototype would report that the host at the IP used by the Ubuntu desktop gateway is either of the 3 Linux versions or Windows 7. Expanding this to a larger NAT environment with a greater variety of operating systems, it is clear that the prototype would eventually, as more traffic is analyzed, treat the IP address as if it could be virtually any operating system. This means that the prototype would eventually be unable to make any decisions, as any alert would be assumed to be valid.

By using the IP, port and protocol however, the prototype would often severely decrease the amount of information about a host, in exchange for much greater precision. By no longer using the IP address alone as identification, the system may even lose all information it previously had available about a target, if that specific port and protocol combination has yet to trigger a PRADS signature.

On the other hand, we will get an opposite effect when the NAT-ed network grows larger and more diverse over a period of time while using the stricter host identification. As more time passes, the prototype will be better equipped to make decisions as the chance of triggering PRADS signatures grows. For these reasons, the stricter host definitions were used in the prototype. Furthermore, this means that the system becomes more dependent on having a reasonably good set of signatures, to be able to identify operating systems at comparable speed as before. Although these scenarios were concerned with operative system detection, the same ideas apply to service detection as well.

IP masquerading

The basic and most common form of NAT is often referred to as masquerading in for example iptables[18], and allows for outgoing connections from client applications to be dynamically NAT-ed. This introduces an additional problem in regards to client applications. Consider a Ubuntu 11.10 host browsing the Internet with the port number `34567` assigned at the gateway. The HTTP

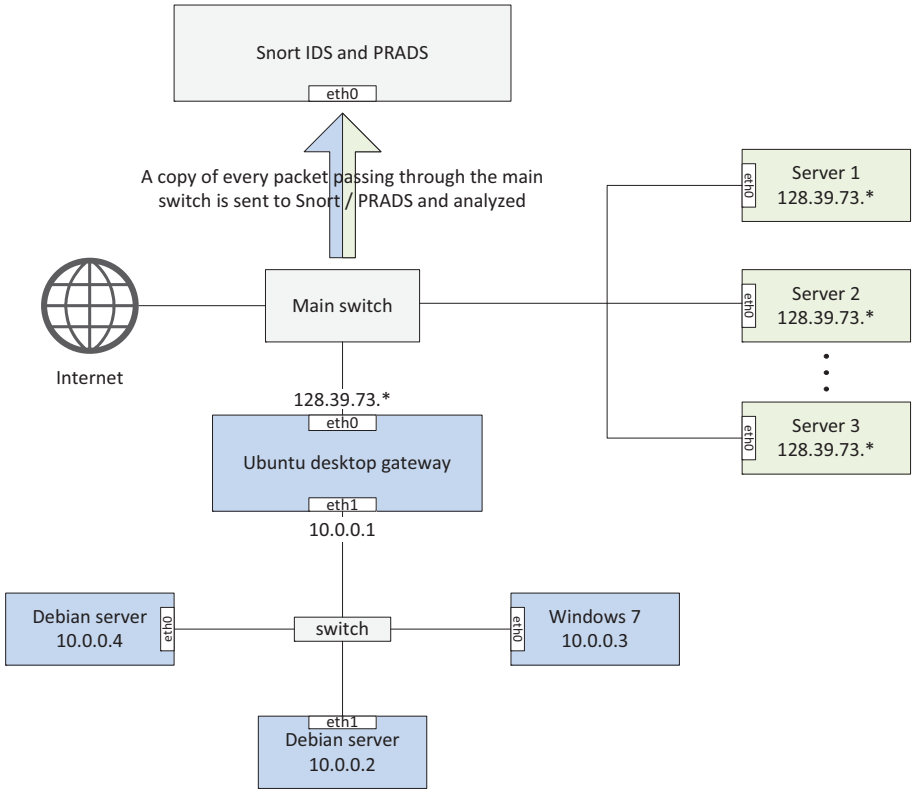

Figure 4.2: A network topology where 4 different hosts share an IP address.

traffic triggers a PRADS signature which registers the IP address, port 34567 and the TCP protocol as a Linux kernel 3.0 machine. Later, when the port 34567 is no longer used and has been released, a Windows 7 machine on the same internal network is assigned the same port and downloads a virus, compromising the host. While Snort notices this and produces an alert, PRADS has no matching signature and still thinks a Linux 3.0 host resides here.

In turn, this would cause the validation program to believe that the alert should be ignored, since the executable file in question does not affect Linux. To deal with such special cases, a timer was introduced to the prototype, which should be set as high as possible, but no higher than the timer used for determining when a socket can be re-used in the NAT device. Alternatively set higher, at a somewhat low risk of real alerts being ignored.

When the prototype looks for host entries in the PRADS database, entries with port numbers above `1023` will be ignored if they are older than the timer permits. This way, we can be reasonably sure that the problem described above will not occur. Port numbers above 1023 were chosen because the operating system generally does not assign the first 1023 ports to clients programs. Ports below this threshold are used by processes that provide services, and should

be NAT-ed with static rules to be accessible.

Allowing aging host information to become outdated will also help with confusion imposed by DHCP (described in section 2.4). Dynamic IP addresses are usually assigned hosts that are not providing services, so the problems caused by DHCP can be alleviated using the same timer solution as for IP masquerading. The aging feature will necessarily reduce the amount of available host information at the benefit of ensuring correctness. This means that it will be even more important that signatures are frequently triggered, to replace any outdated information.

With a larger number of machines behind a NAT-ed network, one can expect the same sockets to be re-used more often. Similarly, with more hosts drawing dynamic IP addresses from an IP range the same IP address can be assumed to be re-used more frequently.

4.2.6 Categorizing alerts

The original plan for the prototype was to simply assign a value to each alert, stating whether it was likely to be a true or false positive. As the system developed however, the fact that alerts cannot always be easily lumped into two categories was recognized.

A simple scoring algorithm was implemented in the prototype based on how sure the program can be that an alert is a true or false positive. The criteria for determining an alerts validity are based on whether the OS, service and versions in the host information table and vulnerability information table match.

- Alerts that are valid against any target (OS, Service and their versions) are assigned a lower score, since such attacks are generally less prone to succeed. SSH brute force attacks would be one example.

- Alerts that are valid against specific targets are assigned a higher score, since they are more prone to have succeeded. A buffer overflow attack exploiting a bug in a specific software version is one such example.

- As targets become more specific, score is increased, while the score will be lower as they become more ambiguous.

- If another OS or service is detected on that IP, port and protocol number than what the vulnerability info defined as applicable, score is greatly reduced

- If no information is available, the score remains unchanged, since the prototype does not know what to make of the alert

Finally the priority assigned to an alert by Snort is used as a weight to produce a final score. This score can then help a busy system administrator determine which alerts should be investigated first. Alerts with very low scores may even be ignored entirely.

4.2.7 Problematic protocols

Since IP addresses are defined at the transport layer of the Internet protocol suite, lower level protocols cannot be handled by the prototype implementation. This minor limitation means that fingerprints made by PRADS on for example the Internet Control Message Protocol (ICMP) cannot be used, and alerts on the ICMP protocol in Snort cannot be validated.

4.3 Prototype program flow

Figure 4.3 illustrates how the prototype works through a simplified program flow chart. The `validateAlert` subroutine is shown in light blue on the right side. See appendix A for the complete source code for the prototype.

4.4 Testing

To determine the viability of the proof of concept introduced in this project, it was necessary to produce some Snort alerts for testing purposes. Real network traffic in a part of an unfiltered network on the HiOA was logged to a file through the `tcpdump` command. Due to the high network load, it was necessary to increase the capture buffer, though even when set to the maximum value a few packets were still dropped.

Over the course of 5 days, roughly 60 GB of network traffic was logged to a PCAP format file. 104960682 packets were logged while 1385 packets were dropped by the kernel due to higher traffic load than the server could handle.

The commands used to collect and analyze the network traffic with Snort and PRADS are shown in code listing 4.13.

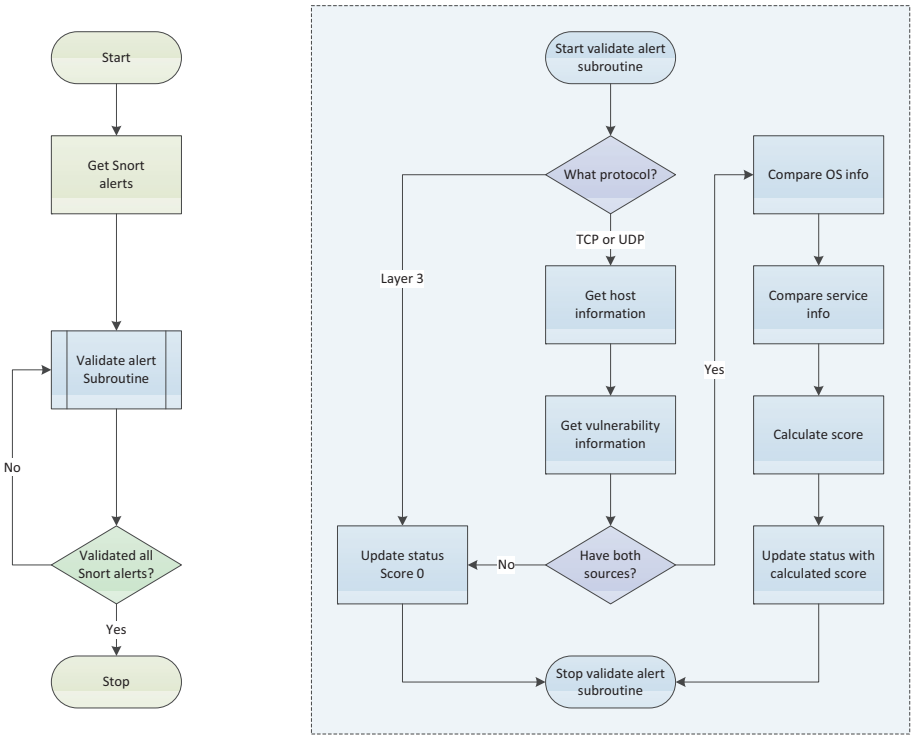

Figure 4.3: A simple program flow chart for the prototype.

```
1 tcpdump -n -s 0 -B 1048576 -i eth1 -w /root/tcpdump.pcap
2 snort -r tcpdump.pcap
3 barnyard2 -d /var/log/snort -f snort.u2 -w /var/log/snort/
    barnyard2.waldo
4 prads -r tcpdump.pcap -l /var/log/snort/prads-assets.log
5 ./prads2db --file /var/log/snort/prads-assets.log
```

Code 4.13: Generating Snort alerts

Three new Snort rules were added to the Snort rule set for testing purposes. These were made to simulate detection of attacks against two specific web-servers, the Apache HTTP server and Internet Information Services 7.5 (IIS). The first rule defined an exploit attempt of an Apache server earlier than version `2.2.22` on any operating system. The second rule was valid against IIS 7.5 on Windows Server 2008 R2 and Windows 7. Finally the third rule would trigger on attacks against Apache version earlier than `2.2.22` on Linux kernels earlier than `3.0`.

This approach allowed the prototype and the system surrounding it (PRADS and Snort), to see a great variety of alerts and hosts. In turn, this made it possible to observe how the prototype behaved in a variety of situations. It was expected that the effectiveness of the prototype's decisions would depend

heavily on the information available on a per-alert basis.

The test results are discussed in chapter 5.

Chapter 5

Test results

The results from testing the prototype implementation are displayed and also explained to some extent in this chapter.

5.1 General testbed overview

The Snort alerts generated by analyzing the tcpdump log file are displayed in table 5.1. The alerts with signature IDs (SID) `9876500`, `9876501` and `9876502` were triggered by custom signatures made purely for testing purposes. These three signatures did not correspond to real attacks, but where triggered using a special TCP payload as defined within these Snort rules.

Signature	SID	Times seen
OpenSSH sshd Identical Blocks DOS attempt	17317	29
Download of executable content - x-header	16313	5
Download of executable content	11192	9
EXPLOIT against IIS 7.5 testrule	9876501	27
EXPLOIT against Apache $<$ 2.2.22 testrule	9876500	6
EXPLOIT against Apache $<$ 2.2.22 on Linux $<$ 3.0 testrule	9876502	10
DOS Apache mod_ssl non-SSL connection to SSL port denial of service attempt	11263	26

Table 5.1: The Snort rules triggered while testing the prototype

The corresponding `vulnerabilityinfo` table entries defining the applicable targets of these alerts are shown in table 5.2.

SID	Operating system	Service
17317	0	openssh:< 4.4
163134	windows	0
11192	windows	0
9876501	windows:2008R2/7	Microsoft-IIS:7.5
9876500	0	apache:< 2.2.22
9876502	Linux:< 3.0	apache:< 2.2.22
11263	0	apache:< 2.0.55

Table 5.2: Vulnerability definitions for the Snort rules

After analyzing the network traffic, PRADS had stored 15683 entries about assets it had detected on 4323 different IP address. Of these, 211 IP addresses resided within the `128.39.73.0/24` subnet.

5.2 Prototype results

5.2.1 Alerts with undetermined statuses

16 out of the 112 alerts processed by the prototype did not have a corresponding host entry on the exact IP, port and protocol combination used to trigger the Snort alert. This meant that the prototype was unable to validate these alerts. These 16 alerts were generated by traffic originating at 15 IP addresses on the `128.39.73.0/24` network. 13 of these were assigned to student virtual machine networks, consisting of a Linux with NAT and one Windows 7 machine using the Linux machine as a gateway. The fact that the prototype was unable to find host information related to these hosts shows that the solution implemented to handle NAT is limiting the the results.

PRADS did however have several entries for these 13 virtualized networks, which showed that the following operating systems might be present: Windows 2000 SP2+, Windows XP SP1+, Windows 98, Linux kernel 2.6, Linux kernel 2.4. The 3 other IP addresses also had a variety of operating systems, including Windows 98 SE, Windows XP, Windows 2000, Windows 7, Windows 2008 R2 and FreeBSD 4.8. This confirms that the NAT solution was necessary, at the very least in this heavily virtualized network.

The source of the problem is that 14 of the these unvalidated alerts referred to clients downloading executable Windows files (.exe) through HTTP. When browsing through different websites, new TCP connections are established at

a high pace, depending on the sites and if solutions like persistent HTTP are used. This means that the executable downloads mentioned above could have been done through a very limited number of packet exchanges, leaving very few chances for PRADS to fingerprint the machine on that exact port.

Having a larger set of well written signatures will presumably help alleviating this problem, similarly to how it would help with the host information aging mentioned in subsection 4.2.5 IP masquerading. Ideally PRADS would be able to quickly produce a fingerprint for every port used.

5.2.2 Alerts with determined statues

The remaining 96 alerts had entries in both the host and the vulnerability information databases. There was much variation in how much information was available for these 96 alerts. The summary below shows what information was available to how many alerts.

- For 25 of the alerts the prototype was unable to determine if there was an operating system match, because their host entries did not contain any operating system information.
- 8 of the alerts had an exact match on the operating system used and the vulnerable system defined in the vulnerability information database.
  - All of these alerts had an exact match on the operating system version as well.
- 4 alerts had an operating system and version mismatch between the host and vulnerability information databases. This means that another operating system was detected than what the attack was aimed at, hence it was very unlikely that the target was vulnerable.
- 59 alerts were defined as valid against “any possible operating system”.

- For 8 of the alerts the prototype was unable to determine if there was a service match, because their host entries did not contain any service information.
- 8 of the alerts where service entries were present did not have a service version listed
- 80 of the alerts had a service match
  - 36 of these alerts had a service version match, while another 36 had a service version mismatch.

- 8 alerts had a service mismatch. This means that another service was detected than what the attack was aimed at, hence it was very unlikely that the target was vulnerable.
- None of these 96 alerts were defined as valid against "any possible service", however 16 alerts were valid against any version of the service they were targeted at.

5.2.3 Investigating the prototype decisions

After the prototype had been applied to the Snort alerts, each of the 112 alerts had received a score in the range `100` to `-200`. A higher score implies that the alert is more relevant, while a negative score implies that the alert is irrelevant and may be ignored. Alerts where the prototype was unable to make a decision, in this case the 16 alerts discussed in subsection 5.2.1, had their scores set to `0`. Refer to appendix B for a complete overview of the scores assigned to the Snort alerts.

An excerpt of the unique alerts and scores assigned by the prototype is shown in table 5.3. The alerts with the highest scores are the ones where all the criteria for the attack have been met, while the lowest scores are assigned to alerts where none of the criteria are met, and instead conflicting operating systems and services are detected. This is the case when an exploit is targeting Windows machine with IIS 7.5, but the host is Linux running Apache 2.

Table 5.3: Excerpt of the prototype results

Signature	Priority	IP	Port	Score
EXPLOIT against IIS 7.5 testrule	1	2150058472	80	100
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058474	80	100
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
OpenSSH sshd Identical Blocks DOS attempt	1	2150058289	22	80
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058471	80	60
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058413	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058419	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058494	443	0
Download of executable content	1	2150058290	55810	0
Download of executable content - x-header	1	2150058320	60028	0
OpenSSH sshd Identical Blocks DOS attempt	1	2150058281	22	-10
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2688863725	80	-120
EXPLOIT against IIS 7.5 testrule	1	2150058474	80	-200

By running the prototype in debugging mode, it was possible to see exactly how the prototype determined what score to assign to each alert. Code listing 5.1 shows the debug mode output for the three IIS 7.5 alerts. In the first alert all

of the host properties can be seen to match the vulnerability definitions. The number `15` is the internal reference to Windows 2008 R2 used for Windows version comparison. A score is then assigned based on the the discoveries, and is then weighted based on the rule priority defined in Snort.

In the second alert, the prototype is able to determine that the server is running an applicable service, but has no information about the operating system. In this special scenario, it would be possible for a human to deduce that the operating system is either Windows 7 or Windows 2008 R2, since IIS 7.5 is only available to these two operating systems. Unfortunately the prototype implementation is not intelligent enough to determine this fact.

Finally the third alert is given the very low score of `-200` because the server is running Linux and Apache, which means that we can sure that it does not run Windows and IIS. If we simply had no host information was available, we could not have been sure whether the alert was valid or not.

```
1  DEBUG: 'windows' matches 'Windows'
2  DEBUG: Versions 15 and 15 are equal
3  DEBUG: 'Microsoft-IIS ' matches 'Microsoft-IIS '
4  DEBUG: Versions 7.5 and 7.5 are equal
5  DEBUG: Score calculator got these: [1] [1] [1] [1]
6  DEBUG: Alert priority: 1
7  DEBUG: '128.39.73.232:80' with sig 'EXPLOIT against IIS 7.5
       testrule' (cid: 1080, prio: 1) got the score 100
8
9  DEBUG: 'Microsoft-IIS ' matches 'Microsoft-IIS '
10 DEBUG: Versions 7.5 and 7.5 are equal
11 DEBUG: Score calculator got these: [0] [0] [1] [1]
12 DEBUG: Alert priority: 1
13 DEBUG: '160.68.205.237:80' with sig 'EXPLOIT against IIS 7.5
       testrule' (cid: 788, prio: 1) got the score 60
14
15 DEBUG: We were looking for windows, but the OS is actually Linux
16 DEBUG: We were looking for Microsoft-IIS , but the service is
       actually Apache
17 DEBUG: Score calculator got these: [-1] [-1] [-1] [-1]
18 DEBUG: Alert priority: 1
19 DEBUG: '128.39.73.234:80' with sig 'EXPLOIT against IIS 7.5
       testrule' (cid: 778, prio: 1) got the score -200
```

Code 5.1: Running the prototype in debugging mode

From table 5.3 it can be seen that an OpenSSH alert was given the rather high score of 80. This alert was generated from real network traffic triggering a real Snort alert. The applicable target for this alert is OpenSSH versions below `4.4`, while the target was actually running version `3.8.1p1`. Since this

alert matches any operating system, the prototype will not give the alert a full `100` score despite being relatively likely to have succeeded. Other OpenSSH alerts were given a much lower score, `-10`, because these were running versions above `4.4`. These alerts were quite unlikely to have succeeded, so it may have been appropriate to assign an even lower score. The scores assigned are easily adjustable according to the system administrator's preferences however.

The `DOS Apache mod ssl non-SSL connection to SSL port denial of service attempt` alerts were real attacks from IP addresses based in Shanghai, China. Since the only information the prototype was able to confirm was that the attack was using TLS, while in other cases no information was available besides the fact that the attack was valid against any operating system, these alerts were given quite low scores.

Finally, the executable file download alerts were completely ignored by the prototype because no host information was available as discussed in subsection 5.2.1.

Chapter 6

Discussion

In this chapter the approach to answering the research questions, the test results and the design are discussed. Future additions to the prototype research in the field is also suggested.

The process of planning, designing and implementing a working prototype has required solutions to several important problems to be found. Designing the system while creating as few limitations as possible has required much planning and many choices had to be done. One of the earliest problems that had to be dealt with was to find the most appropriate way to gather the necessary network information to provide host awareness to the prototype. A signature based passive fingerprinting solution was implemented through PRADS, which was a quite different approach to what has previously been done in similar systems. As a part of the passive fingerprinting solution, NAT also had to be handled in a way that produced correct results without limiting the available host information more than necessary.

A significant amount of time was spent on an attempt to find a way to automatically gather all of the available vulnerability information from external Internet sources. This proved excessively problematic due to issues such as vulnerability descriptions being written in arbitrary ways meant to be read by humans. This problem was eventually left unsolved in favor of implementing the prototype to a level where it could be tested. This was considered to be a higher priority because unlike automatically gathering vulnerability information, testing the prototype was vital to answering the problem statements.

Another major problem was to help the prototype understand the data it was provided with, so that the information could be compared and a decision be

made. Parts of the PRADS fingerprints had verbose information presumably designed to be human readable. Eventually slight modifications were made to the PRADS fingerprints to allow easier information extraction in the prototype.

In general, reports of previous work was lacking detailed information on selected interesting topics such as how external vulnerability information was parsed and made readable by their implementations. Having more information on their solutions to such problems would mean that these problems could be solved more quickly in this project, allowing time to be spent more efficiently. Furthermore, the commercial solutions that claimed to have similar features had very limited information available on the internal mechanics dealing with network and context awareness. Furthermore, Sourcefire for example required registration and signing up for newsletters to gain access to white papers.

Snort was configured to a minimal level that one might expect from a real world system where only limited information about the network was available. Rules that were producing excessive amounts of general informational alerts were removed. The only possible validation the planned prototype would be able to perform on alerts such as for example SSH brute force attacks would be to check if the target was using the default port 22. This type of alert filtering is already easily done by configuring the network's SSH ports in Snort. In retrospect, it could have been interesting to see how well the prototype had performed in comparison to a completely un-configured IDS, and also in different networks.

6.1 Discussion on the test results

6.1.1 Possible error sources

The signatures used for service detection in PRADS rely on matching regular expression patterns with known text strings reported by the servers. This means that we have to trust these servers to report correct information about their services. Often these text strings are easily modified in the service's configuration files. Usually these are only modified for security reasons, where one might not want the service version and patch levels to be publicly available. These strings could just as easily be changed by a mischievous service operator to report incorrect information. As an example, an Apache 2 HTTP server could be configured to report that it is in fact an OpenSSH server. This

means that there was a chance that the final scores in some of the alerts with service definitions would be incorrectly assigned. Through manual inspection of these alerts, it was shown that no such cases existed in the test data. It was not possible to determine if the service versions reported by some of the services had been tampered with however. In contrast to the regular expression pattern matching, the PRADS operating system signatures and a portion of the signatures used in Snort analyze the characteristics of the datagram as well as binary or hexadecimal content. This makes it much more difficult to tamper with the information.

Using `tcpdump` and logging data for almost a week before analyzing it with Snort and PRADS can be assumed to have affected the test results. When testing the prototype PRADS would have host entries dating back several days which could then be applied even to the first alerts generated. If the testing was done on live network traffic and the validation was done continuously as new alerts were generated, the results may have been worse, depending on how quickly PRADS would detect the relevant assets. Despite this, using tcpdump was justified by the fact that it allowed experiments to be redone without having to spend several days on collecting new data had something gone wrong. The tcpdump approach used here may be compared to a system where testing only began after the system had been running for a few days. Dynamic IPs did not cause errors, as the `128.39.73.0/24` network did not use DHCP. Furthermore the "test rule" alerts showing attacks against targets outside the local network were created near the end of the packet logging session. In retrospect it could have been interesting to compare the results of tests where the system started from scratch with no host information with results where the system has been running for a while to get a decent network overview.

The testing was done in a real environment using real network traffic. A portion of the alerts were made from custom made signatures with the sole purpose of generating a variety of alerts. The results are reproduceable in that all the steps necessary to set up a similar environment, and the source code for the prototype is provided. Additionally all tools used throughout the project were free and open source. Despite the large amount of network traffic inspected, relatively few alerts were produced by the IDS. This is likely because basic configuration was done to eliminate general informational alerts such as warnings about SSH brute force attempts. Furthermore, the HiOA network blocked incoming traffic on a few common ports used by remote desktop services such as VNC and RDP.

6.1.2 Expectations

It was expected that the prototype would be able to determine the validity of a portion of the Snort alerts, and that the available host information for each target would be the limiting factor in the decisions. The results of the testing show that almost all of the alerts were assigned an appropriate score. The top scores of 80 to 100 were all assigned to alerts that could be assumed to have been completely valid, since all of the available requirements were met. The alerts with score 60 all had a match on the service and service version, but the prototype was unable to confirm the operating system due to lack of host information.

Of the 29 OpenSSH attacks, only one OpenSSH server was determined to be vulnerable due to a service version match. The remaining 28 attacks were assigned scores below the default score of `0`. A few alerts with extremely low chances of success were given scores as low as `-120` and `-200`.

The `DOS Apache mod ssl non-SSL connection to SSL port denial of service attempt` rules were at times assigned slightly higher scores than desired. The only fact that the prototype was able to confirm on these alerts was that the traffic was using SSL/TLS encryption. In reality this specific denial of service attack also required Apache versions 2.0 to 2.0.55 to be installed and configured in a very specific way. The prototype knows nothing of the files on the target system however so these scores were to be expected.

Overall the results show that the prototype was very effective in categorizing the alerts with appropriate scores.

6.1.3 Comparison to related work

The end results of a prototype such as the one implemented in this project will depend heavily on the network on which it is tested. This makes it difficult to compare the results with reports of similar implementations. A network consisting entirely of servers where all services are known to the fingerprinting system would for example give a completely different result than a network with many unknowns or several possible configurations detected. Similarly the type and number of alerts would also have a large impact, since it is usually much easier to determine the success of a targeted attack than a general attack.

In the related work discussed in section 2.5, several different numbers indicating how many false positives or even false negatives were removed by their

systems are reported, however in general they are not directly comparable. Furthermore the prototype implemented in this report does not simply categorize alerts as false or true positives. By looking at the number of alerts that were assigned a score, it is clear that a large number of alerts were categorized with the help of the prototype. The end result of both the prototype introduced in this report and the related work were similar in that all of the systems were able to categorize the alerts to some extent based on the available information.

6.2 The resulting prototype

The prototype implemented in this project can be seen as a more realistic implementation than some of the public research done in earlier years. Most of the previous approaches have either had terrible scalability due to using active scanning techniques, which in turn limited their alert validations to a few ports on the target machines, unrealistically assumed authenticated access to each of the hosts on the system, or assumed that each host has all their relevant configurations defined in a database. The prototype introduced in this project does not have these problems or assumptions.

None of these earlier approaches have mentioned taking into consideration important networking technologies such as NAT and dynamic IP addresses through DHCP. This means that their solutions are prone to making errors in virtualized networks, or rapidly changing networks where hosts are leaving and joining the network. The prototype implemented here was designed with both of these networking technologies in mind. Being able to handle NAT has been extremely important for many years now due to the popularity of virtualization. Furthermore, being able to handle dynamic IP addresses is important if one wants to have an IDS in a network where for example people bring their own mobile devices, which is also becoming increasingly common.

Stealthiness, meaning it's harder for an intruder to detect and know about the system is also a key benefit of using passive fingerprinting, and compliments an IDS with mirrored traffic very well. For an attacker, this could mean that he does not know which IDS evasion method he should use, or if any IDS is even present at all.

Another major difference between the prototype developed in this project and previous similar implementations is that this prototype assigns scores rather than simply using two predefined categories. These scores relate to a degree of likeliness that an attack was successful as well as the severity of the attack. In a busy system where the administrator only has time to investigate a small

subset of the alerts, it is extremely helpful knowing which alerts were most likely to have succeeded, and also which alerts had too many unknowns to be categorized.

One side effect of the current prototype implementation is that it might hide outgoing attacks from the network if the alerts from these were considered to be unsuccessful by the system. System administrators would probably like to know about attacks done from the network they are responsible for. A simple solution to this could be to configure PRADS to only log host information for the network it resides in, at the expense of not being able to determine if outgoing attacks could have been successful. On the other hand, knowing if an outgoing attack was successful may not be as relevant to the system administrator. Systems without active or passive fingerprinting do not have to consider this issue, since they are unable to gather host information about outside sources. It should also be noted that active scanning of hosts outside the local network may have legal implications.

6.3 Future work

6.3.1 Future work in this topic

The definitions in the existing vulnerability information databases are either meant to be human readable and therefore extremely difficult to interpret automatically at times, or have a somewhat strict but verbose format which is only accessible by downloading individual HTML pages. A publicly available, downloadable database with vulnerability definitions following a strict format needs to be developed for usage in this and other network security systems.

PRADS fingerprints should be extended with additional signatures with machine readability in mind. A strict format for defining version ranges and multiple possibilities should be required rather than the currently existing question marks, comments and other information meant to be human readable. For the most part the signatures used in PRADS appear to stem from old p0f or PADS signatures. Considering the relatively small communities surrounding these projects and probably limited usage as parts of other systems, it is understandable that stricter formatting has not been prioritized earlier. Furthermore, it may be beneficial if service signatures based on binary and hexadecimal payloads were introduced in PRADS. That way the system cannot be tricked as easily by mischievous administrators.

The host and vulnerability information gathered by such systems could provide useful information to IDS front ends such as BASE. Having all the related information in one place would be convenient for investigating the alerts manually.

Sourcefire 3D supposedly provides real-time user awareness through a module, which is able to link IP addresses to active directory and LDAP users. This means that the system administrator does not have to find this information manually. Such a feature could also be added to the prototype implemented here. More importantly however, with the agent solution proposed in section 6.3.2 an idea worth exploring could be reporting of which system user was responsible for the offending process in a Snort alert.

6.3.2 Prototype improvements

Since the main goal of the prototype developed in this project was to provide answers to the research questions rather than implementing a production ready system, some additional features should be implemented for it to be usable in a real world solution. Firstly the system does not currently remove old PRADS data, meaning the database will eventually grow very large. An intelligent solution for data aging should therefore be implemented. This solution could for example consider the entry's age, how frequently the asset has been seen with same configuration and maybe even consider how sane the configuration on the system appears to be. Having a Linux server with IIS 7.5 would for example be difficult to achieve, while windows machines with apache HTTP servers are relatively rare in comparison to IIS.

Better support for various combinations of meta characters such as $<, >, >=$, $<=, +, -$, for version numbers would be convenient for both host and vulnerability definitions. This way version number ranges could be defined. This was not considered a high priority in the prototype since the alerts did not require them. A real implementation would require significant testing on all the possible combinations of meta characters.

The solution shown in the prototype should be better integrated into Snort, so that the solution works in IPS (intrusion prevention) mode as well. This will require heavy optimization and quite likely an implementation written in C. The database information used to make decisions should be kept in memory for quicker access, since reading from a hard drive is much too slow when running an IDS in inline mode.

The score calculations and its configuration possibilities could be made much more feature rich and advanced. Although the testing done in this project showed that the assigned scores were generally quite correct, the current method is very simple and could probably benefit from a more complex solution.

The ability to define single IP addresses or subnets that do or do not have NAT or DHCP would be useful. That way the prototype could work more efficiently by making use of all the relevant host information which would otherwise go to waste.

Agent based solution

Having an agent based solution to compliment the passive fingerprinting system for host information gathering was an attractive prospect given much thought earlier in this project. The idea was later dropped due to the extra amount of work necessary to handle NAT in such a solution. This feature was not considered important enough in regards to answering the research questions to justify the time investment. Nevertheless, the idea was explored thoroughly and appropriate solution was found albeit not implemented.

On Ubuntu machines which the agent was developed for the `ss -tulpn` command, an improvement to the popular `netstat` command provided all the necessary process information about services listening for incoming traffic. Together with the `dpkg -l` command and some regular expressions it was then possible to extract exact version numbers for several services.

In networks with NAT, it would then be necessary to communicate the information to an agent on that machine, which could translate the port numbers into their correct values as they would be seen from the intrusion detection system. An example of this setup is shown in figure 6.1. The servers without NAT communicate their information directly to the IDS machine, while the machines with local IP addresses have to communicate their information to the agent on their gateway.

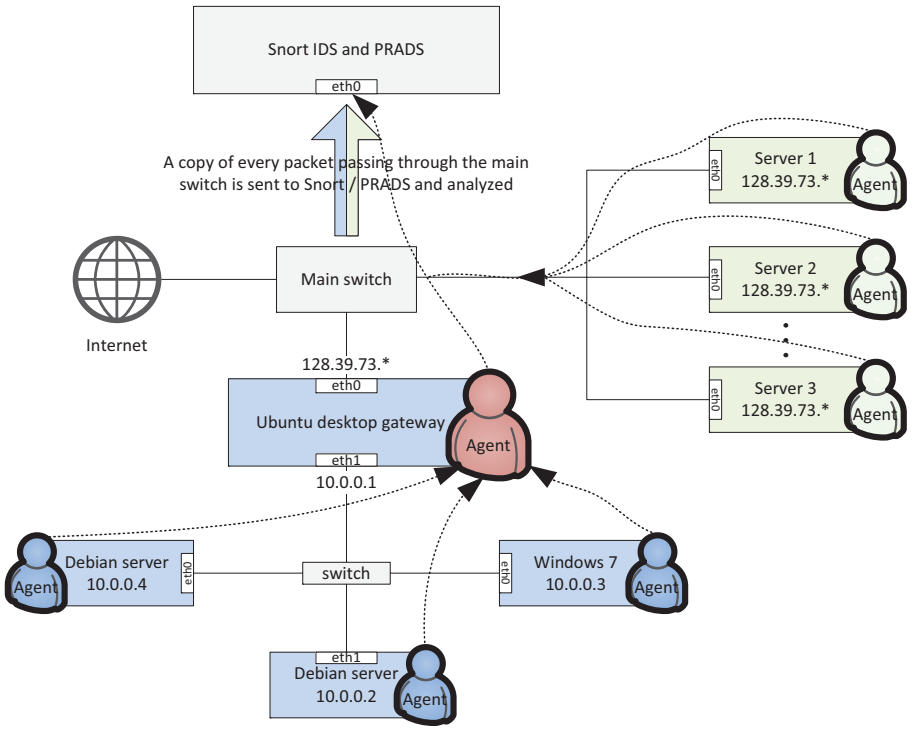

Figure 6.1: An agent keeps track of the NAT tables for its local network.

Chapter 7

Conclusion

7.1 Conclusion

The main goal in this thesis has been to design, implement and test a prototype which introduced network and context awareness to an intrusion detection system (IDS), with the purpose of reducing the number of false alarms reported. To achieve this goal, three relevant research questions were investigated.

I. In what way can existing open source software tools best be used for collecting host information relevant to intrusion detection in a network?

Passive fingerprinting was determined to be the best solution for the prototype. Key benefits included excellent scalability, stealthiness by not leaving any network footprints, quick asset detection assuming decent signatures are used, no authenticated access or control necessary on the hosts and inherent strengths in handling mobile devices and dynamic networks. PRADS was chosen because it appeared to be the most advanced and up to date open source software tool for passive fingerprinting at the time.

II. How can the collected host information be applied to the alerts produced by signature based IDS such as Snort?

Host information can be compared to online vulnerability definitions which are referenced in most Snort rules. Several information sources were considered, but none were able to provide an API or downloadable database with machine readable content. As a temporary solution, a small portion of the

vulnerability definitions were manually added to a local database for testing purposes. It is hoped that the online sources will be improved in the future.

III. What are the benefits and drawbacks of introducing network awareness in an IDS, effectively making it a target based IDS?

By implementing a prototype it was possible to demonstrate the possibility and effectiveness of classifying Snort alerts according to their relevance, by providing the IDS with network awareness. Making host information available to Snort has helped the IDS on its way towards becoming a "next generation" IDS. Now that fundamental requirements for a "next generation" system has been provided, a logical next step could for example be the addition of user awareness. On the downside, relying on another layer of signature based decisions meant that a possibility of additional errors was introduced, albeit rarely.

The final verdict was that the prototype was able to assign appropriate scores to Snort alerts generated from real network traffic, based on the information it had available on each individual incident.

7.2 Additional contributions to the field

There has been significant research in anomaly based IDS compared to signature based IDS for many years. This does not reflect the real world where signature based systems completely dominate the field. This report adds to the signature based IDS research field with new insight into network and context awareness.

Problems caused by Network Address Translation and DHCP which have been entirely ignored in previous work were discussed and solved. The prototype was designed with these two technologies in mind throughout the whole process. This means that the solution can be applied to virtualized networks and environments with mobile devices, both of which are becoming increasingly popular.

Previous similar work has been largely focused on providing IDS network awareness though active scanning, configuration databases or agent based solutions. Furthermore, much of the previous work has been focused on HTTP traffic. In contrast, the prototype developed here allows for passive detection of operating systems as well as services and their versions by comparing TCP and UDP packets to known signatures.

Bibliography

[1] C.V.N. Index. Forecast and methodology, 2010–2015. *San Jose, CA*, 2011.

[2] M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, M.K. Low, D. Mazurek, D. McKinney, et al. Symantec internet security threat report trends for 2010. *Volume XVI*, 2011.

[3] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Trans. Inf. Syst. Secur.*, 6(4):443–471, November 2003.

[4] C.Y. Ho, Y.C. Lai, I.W. Chen, F.Y. Wang, and W.H. Tai. Statistical analysis of false positives and false negatives from real traffic with intrusion detection/prevention systems. *Communications Magazine, IEEE*, 50(3):146 –154, march 2012.

[5] Cisco Threat Response. `http://www.cisco.com/en/US/products/sw/secursw/ps5054/index.html`. Visited march 2012.

[6] CTO of Sourcefire M. Roesch. Seclists.org mailing lists. `http://seclists.org/focus-ids/2004/Jan/56`. Visited march 2012.

[7] E. Hughes and A. Somayaji. Towards network awareness. In *Proceedings of the 19th Large Installation System Administration Conference (LISA 05)*, pages 113–124, 2005.

[8] Sourcefire 3d rua. `http://www.sourcefire.com/security-technologies/cyber-security-products/3d-system/user-awareness`. Visited may 2012. Datasheet and technology brief available.

[9] Sourcefire 3d rna. `http://www.sourcefire.com/security-technologies/cyber-security-products/3d-system/network-awareness`. Visited may 2012. Datasheet and technology brief available.

[10] C. Young. Cisco keynote at RSA Conference 2012. http://www.cisco.com/web/learning/le21/spotlight/rsa/2012/details.html?&_PRIORITY_CODE=cdcsec. Visited april 2012.

[11] R. Koch. Towards next-generation intrusion detection. In *Cyber Conflict (ICCC), 2011 3rd International Conference on*, pages 1–18. IEEE, 2011.

[12] Nmap home page. http://nmap.org/. Visited april 2012.

[13] The Hackers Choice (THC) Amap home page. http://www.thc.org/. Visited april 2012.

[14] D. Lee, J. Rowe, C. Ko, and K. Levitt. Detecting and defending against web-server fingerprinting. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 321 – 330, 2002.

[15] PRADS at github.net. https://github.com/gamelinux/prads. Visited may 2012.

[16] p0fv3 home page. http://lcamtuf.coredump.cx/p0f3/. Visited may 2012.

[17] Ettercap home page. http://ettercap.sourceforge.net/. Visited april 2012.

[18] Iptables project. http://www.netfilter.org/projects/iptables/index.html. Visited may 2012.

[19] Snort home page. http://www.snort.org/. Visited may 2012.

[20] The Bro network security monitor. http://bro-ids.org/. Visited may 2012.

[21] Hogger - nmap to host attribute table. http://code.google.com/p/hogger/. Visited may 2012.

[22] prads2snort.pl - prads to host attribute table. https://github.com/gamelinux/prads/blob/master/tools/prads2snort. Visited may 2012.

[23] L.C. Miller. *Next-Generation Firewalls for Dummies*. Wiley Publishing, 2011.

[24] Cisco Systems. Cisco asa cx delivers context-aware security. http://www.cisco.com/en/US/prod/collateral/vpndevc/ps6032/ps6094/ps6120/white_paper_c11-700240.pdf, February 2012. Visited may 2012.

[25] C. Kruegel and W. Robertson. Alert verification: Determining the success of intrusion attempts. In *Proc. First Workshop the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA 2004)*, 2004.

[26] Nessus vulnerability scanner home page. `http://www.tenable.com/products/nessus`. Visited april 2012.

[27] E. Karaarslan, T. Tuglular, and H. Sengonca. Does network awareness make a difference in the intrusion detection of web attacks? In *ICHIT 2006*, volume 000, pages 1–10, 2006.

[28] L. Pimenidis, B. Westermann, and V. Wetzelaer. A context aware network-IDS. In *Proceedings of The 13th Nordic Workshop on Secure IT Systems NordSec 2008*, page 3, 2008.

[29] M. Sourour, B. Adel, and A. Tarek. Environmental awareness intrusion detection and prevention system toward reducing false positives and false negatives. In *Computational Intelligence in Cyber Security, 2009. CICS '09. IEEE Symposium on*, pages 107 –114, 30 2009-april 2 2009.

[30] M. Wierbosch. Reducing false positives by passively fingerprinting the web server. In *15th TSConIT*, 2011.

[31] S.O. Al-Mamory and H.L. Zhang. A survey on ids alerts processing techniques. In *Proceedings of the 6th WSEAS international conference on Information security and privacy*, pages 69–78. World Scientific and Engineering Academy and Society (WSEAS), 2007.

[32] G.P. Spathoulas and S.K. Katsikas. Reducing false positives in intrusion detection systems. *Computers and Security*, 29(1):35 – 44, 2010.

[33] J.T. Rødfoss. Comparison of open source network intrusion detection systems. Master's thesis, University of Oslo, 2011. `http://urn.nb.no/URN:NBN:no-29859`.

[34] Cfengine community edition home page. `http://cfengine.com/community`. Visited may 2012.

[35] Puppet labs home page. `http://puppetlabs.com/`. Visited may 2012.

[36] Preboot Execution Environment (PXE) specification v2.1. `http://www.pix.net/software/pxeboot/archive/pxespec.pdf`. Visited may 2012.

[37] Verizon databreach report 2012. http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2012_en_xg.pdf. Visited may 2012.

[38] CSI security survey 2010/2011. http://gocsi.com/survey. Visited may 2012.

[39] SinFP at sourceforge.net. http://sourceforge.net/projects/sinfp/. Visited may 2012.

[40] Common Vulnerability Database. http://cve.mitre.org/index.html. Visited may 2012.

[41] Securityfocus Bugtraq. http://www.securityfocus.com/. Visited may 2012.

[42] Comprehensive perl archive network. http://www.cpan.org/.

[43] Microsoft technet security bulletins. http://technet.microsoft.com/en-us/security/bulletin/. Visited may 2012.

[44] Basic Analysis and Security Engine - BASE. http://base.secureideas.net/. Visited may 2012.

Appendices

Appendix A

validator.pl

```
#!/usr/bin/perl -w

# ./validator.pl -h
# version 1.0

# Required packages
use Getopt::Std;
use strict "vars";
use DBI;
use Socket; # Used for IP address conversion
use Sort::Versions; # Simple version number comparison,
# http://search.cpan.org/~edavis/Sort-Versions-1.5/Versions.pm

# Global variables
my $VERBOSE = 0;
my $DEBUG = 0;

# Handle flags and arguments
# Example: c == "-c", "c: == "-c argument"
my $opt_string = 'vdhS:H:P:D:u:p:';
getopts( "$opt_string",\my %opt ) or usage() and exit 1;

# Note: The DBI module cannot execute queries over multiple databases,
# tables that must be joined for example should reside within the same db.

# Default variable values
my $snortserver = "mysql";
my $snorthost = "localhost";
my $snortport = "3306";
my $snortdb = "snort";
my $snortuser = "snort";
my $snortpw = "";

my $hostinfoserver = "mysql";
my $hostinfohost = "localhost";
my $hostinfoport = "3306";
my $hostinfodb = "prads";
my $hostinfouser = "prads";
my $hostinfopw = "";
```

```
40
41 my $siginfoserver = "mysql";
42 my $siginfohost = "localhost";
43 my $siginfoport = "3306";
44 my $siginfodb = "snort";
45 my $siginfouser = "snort";
46 my $siginfopw = "";
47
48 my $validationserver = "mysql";
49 my $validationhost = "localhost";
50 my $validationport = "3306";
51 my $validationdb = "snort";
52 my $validationuser = "snort";
53 my $validationpw = "";
54
55 # Time in seconds before entries with port > 1023 are outdated in PRADS
56 my $nat_lifespan = 300;
57
58 # Scores for the alert 'categorizing'
59 # note: if a conflicting OS/service is detected, a negative score * 2 is used
60 # note: if any target is applicable, half score is used
61 my @score = (
62     20, # The operating systems match
63     20, # The operating system versions match
64     30, # The services match
65     30, # The service versions match
66 );
67
68 # Print help message if -h is invoked
69 if ( $opt{'h'} ){
70     usage();
71     exit 0;
72 }
73
74 # Handle other user input
75 $VERBOSE = 1 if $opt{'v'};
76 $DEBUG = 1 if $opt{'d'};
77
78 ##### Main script content
79 #
80 verbose("Verbose is enabled\n");
81 debug("Debug is enabled\n");
82
83 # Hardcoded indexed array for known named version comparison, starting at win 95
84 my @windows = ("95", "NT4.0", "98", "98SE", "2000", "Me", "XP", "XP64", "2003", "XPPro64",
       "WinFLP", "Vista", "HomeServer", "2008", "7", "2008R2", "HomeServer2011", "8");
85 my %index;
86 @index{@windows} = (0..$#windows);
87
88 my $counter = 0;
89 verbose("#### VALIDATING ####\n\n");
90
91 getAlerts();
92 verbose("validated $counter alerts\n");
93
94 # Fetches all the alerts which have not yet been validated against the host information
95 sub getAlerts {
96     my $snortdsn = "dbi:$snortserver:$snortdb:$snorthost:$snortport";
97     my $snortdbh = DBI->connect($snortdsn, $snortuser, $snortpw) or die "Error: Could not
          connect to the Snort database:\n" . $DBI::errstr . "\n";
```

```
98
99      # Get all the alerts that have not yet been validated
100     # Think the acid_event table might actually belong to BASE? If so should probably use
            a more complex query to join the various SNORT tables to get the same result
101     my $query = "SELECT sig_name, sig_sid, sig_priority, ip_dst, tcp_dport, udp_dport,
            event.sid, event.cid FROM event \
102                     LEFT JOIN iphdr ON (event.sid = iphdr.sid AND event.cid = iphdr.cid) \
103                     LEFT JOIN tcphdr ON (event.sid = tcphdr.sid AND event.cid = tcphdr.cid) \
104                     LEFT JOIN udphdr ON (event.sid = udphdr.sid AND event.cid = udphdr.cid) \
105                     LEFT JOIN signature ON (event.signature = signature.sig_id) \
106                     LEFT JOIN validation ON (event.cid = validation.cid) \
107                     WHERE validation.status is null";
108
109     my $result = $snortdbh->prepare($query);
110     $result->execute or die "SQL error: $DBI::errstr\n";
111
112     while (my @row = $result->fetchrow_array) {
113   validateAlert(@row);
114   $counter++;
115     }
116
117     # Done, disconnect from the database
118     $snortdbh->disconnect or warn $snortdbh->errstr;
119 }
120
121 # Validates an alert against the information collected for a given host
122 sub validateAlert {
123     my $proto = 0;
124     my $dport = 0;
125     if ($_[4]) {
126   # TCP
127   $dport = $_[4];
128   $proto = 6;
129     } elsif ($_[5]) {
130   # UDP
131   $dport = $_[5];
132   $proto = 17;
133     } else {
134   # Some unknown protocol or layer 3 datagram - Likely ICMP
135   updateStatus($_[6], $_[7], 0);
136   warn "Unknown protocol: cid: $_[7] - sig: $_[0]\n";
137   return;
138     }
139
140     my $target = decimalToIp($_[3]);
141
142     # Fetch the host information and signature information
143     my @hostinfo = getHostInfo($target, $dport, $proto);
144     my ($os, $service) = getSigInfo($_[1]);
145
146     # No point in trying to compare information unless we have info from both sources
147     if (!@hostinfo) {
148   updateStatus($_[6], $_[7], 0);
149   return;
150     }
151     if (($os eq "") && ($service eq "")) {
152   updateStatus($_[6], $_[7], 0);
153   return;
154     }
155
```

```
156     # We have the necessary data, proceed with the comparison
157     my ($os_match, $os_v_match) = compareOS($os, @hostinfo);
158     my ($service_match, $service_v_match) = compareService($service, @hostinfo);
159     my $score = categorize($os_match, $os_v_match, $service_match, $service_v_match, $_
            [2]);
160     debug("'$target:$dport' with sig '$_[0]' (cid: $_[7], prio: $_[2]) got the score
            $score\n");
161     updateStatus($_[6], $_[7], $score);
162 }
163
164 # Assigns a score based on the configured weights
165 # Returns the score assigned to the current alert
166 sub categorize() {
167     my $tmp = 0;
168
169     debug("Score calculator got these: ");
170     for (my $i = 0; $i < 4; $i++) {
171   print "[$_[$i]] " if $DEBUG;
172   if ($_[$i] == 1) {
173       $tmp += $score[$i];
174   } elsif ($_[$i] == -1) {
175       $tmp -= 2 * $score[$i];
176   } elsif ($_[$i] == 2) {
177       $tmp += ($score[$i]/2);
178   }
179     }
180     print "\n" if $DEBUG;
181     debug("Alert priority: $_[4]\n");
182     if ($tmp > 0) {
183   $tmp /= $_[4];
184     } else {
185   $tmp *= $_[4];
186     }
187
188     return $tmp;
189 }
190
191 # Inserts the final verdict into the database for each alert validated
192 sub updateStatus() {
193     my $validationdsn = "dbi:$validationserver:$validationdb:$validationhost:
            $validationport";
194     my $validationdbh = DBI->connect($validationdsn, $validationuser, $validationpw) or
            die "Error: Could not connect to the validation database:\n" . $DBI::errstr . "\n"
            ;
195
196     my $now = time();
197     my $query = "INSERT INTO validation (sid, cid, status, timestamp) VALUES ($_[0], $_
            [1], $_[2], FROM_UNIXTIME($now)) \
198                 ON DUPLICATE KEY UPDATE status=$_[2], timestamp=FROM_UNIXTIME($now)";
199
200     verbose("Insert query: $query\n");
201     my $result = $validationdbh->prepare($query);
202     $result->execute or die "SQL error: $DBI::errstr\n";
203 }
204
205 # Compares host operating system with the vulnerability definitions
206 # Returns two flags indicating the result of the OS and version comparison
207 sub compareOS {
208     my ($osstring, @hostinfo) = @_;
209
```

```
210     # 0 = no match, 1 = match, -1 there is another OS there, 2 = any
211     my $os_match = 0;
212     my $os_v_match = 0;
213
214     if ($osstring eq 0) {
215   # This alert is valid against any OS
216   $os_match = 2;
217   $os_v_match = 2;
218     } else {
219   # Need to investigate further
220
221   # $os format: OS1:(versions)!OS2:(versions)
222         my @systems = split('!',$osstring);
223
224   # For each OS type, could be one or several, but usually only one.
225   foreach (@systems) {
226       # First element contains the system, following elements contain versions if
            applicable
227       my @system = split(':',$_);
228       my $matchedos = $system[0];
229
230       # Check if this OS matches any of several possibly present hostinfo entries
231       foreach (@hostinfo) {
232     my $entryOS = 0;
233     my @entryversions;
234
235     # The OS name can hold anything except from the meta character ':'
236     if ($_ =~ /{([^:]+):?(.*)?}/) {
237         $entryOS = $1;
238         @entryversions = split('/',$2);
239     }
240
241     # Should be exactly same, otherwise 'linux' might match a service like 'service-for-
          linux'
242     if (lc($entryOS) eq lc($matchedos)) {
243         # We have a match on the OS
244         $os_match = 1;
245         debug("'$matchedos' matches '$entryOS'\n");
246
247         # Proceed to check all versions if there is any
248         if (scalar(@system) > 1) {
249       my @versions = split('/',$system[1]);
250       # @versions holds the vulnerability DB versions
251       # @entryversions holds the host info DB versions
252
253       foreach my $vdb (@versions) {
254           foreach my $hdb (@entryversions) {
255         # Decide how to compare versions
256         # Only supports '1.3.2', '<1.2.3', '>3.2.1' at the moment
257
258         # Translating named versions to numeric for comparison
259         if (lc($entryOS) eq "windows") {
260             if ( $vdb =~ /[<>]?([\S]+)(?:\s[A-Za-z0-9+-]+)?/) {
261           my $replace = $1;
262                 my $index = $index{$replace};
263           $vdb =~ s/$replace/$index/;
264           $hdb =~ s/$replace/$index/;
265             } else {
266           warn "Invalid OS version formatting for $entryOS:$hdb, comparison failed.\n";
267             }
```

```
268         }
269
270         # The versioncmp method works just like cmp, but for common version patterns
271         if ($vdb =~ /([<>])/) {
272             my $meta = $1;
273             if ($meta eq '<') {
274           $vdb =~ s/<//;
275           my $res = versioncmp ($vdb, $hdb);
276           if ($res == 1) {
277               $os_v_match = 1;
278               debug("Version $hdb was smaller than $vdb, version is applicable\n");
279           } elsif (($res == -1) && ($os_v_match == 0)) {
280               $os_v_match = -1;
281               debug("Version $hdb was greater than $vdb, version not applicable\n");
282           }
283             } elsif ($meta eq '>') {
284           $vdb =~ s/>//;
285           my $res = versioncmp ($vdb, $hdb);
286
287           if ($res == -1) {
288               $os_v_match = 1;
289               debug("Version $hdb was greater than $vdb, version is applicable\n") if (
                      $res == -1);
290           } elsif (($res == 1) && ($os_v_match == 0)) {
291               $os_v_match = -1;
292               debug("Version $hdb was greater than $vdb, version not applicable\n") if (
                      $res == 1);
293           }
294             }
295
296         } else {
297             # No meta character, just check if versions equal
298             my $res = versioncmp ($vdb, $hdb);
299             $os_v_match = 1 if ($res == 0);
300             debug("Versions $vdb and $hdb are equal\n") if ($res == 0);
301         }
302           }
303       }
304         } else {
305       # All OS versions are applicable
306       $os_v_match = 2;
307         }
308     } else {
309         # If the OS entry from PRADS fingerprint has a trailing whitespace, it's actually
              a service entry
310         if (($entryOS !~ /\s$/) && ($entryOS)) {
311       # If we get there it means that another OS has been detected
312       debug("We were looking for $matchedos, but the OS is actually $entryOS\n");
313       $os_match = -1;
314       $os_v_match = -1;
315         }
316     }
317       }
318   }
319     }
320     return ($os_match, $os_v_match);
321 }
322
323 # Compares host service with the vulnerability definitions
324 # Returns two flags indicating the result of the service and version comparison
```

```
sub compareService {
    my ($servicestring, @hostinfo) = @_;

    # 0 = no match, 1 = match, -1 = another service detected, 2 = any
    my $service_match = 0;
    my $service_v_match = 0;

    if ($servicestring eq 0) {
  # This alert is valid against any service
  $service_match = 2;
  $service_v_match = 2;
    } else {
  # Need to investigate further

  # $service format: srv1:vers/ions!srv2:vers/ions
        my @services = split('!',$servicestring);

  # For each service type, could be one or several, but usually only one.
  foreach (@services) {
      # First element contains the service, following elements contain versions if applicable
      my @service = split(':',$_);
      my $matchedservice = $service[0];

      # The services have a space after the name in the modified PRADS fingerprints,
      # which allows us to distinguish them from OS entries for now
      # Concatinating a space to $entryservice rather than removing one from $matchedservice
      # for later comparison. Optimally PRADS would separate them in the log file
      $matchedservice .= " ";

      # Check if this service matches any of several possibly present hostinfo entries
      foreach (@hostinfo) {
    my $entryservice = 0;
    my @entryversions;

    # The service name can hold anything except from the meta character ':'
    if ($_ =~ /{([^:]+):?(.*)?}/) {
        $entryservice = $1;
        @entryversions = split('/',$2);
    }

    # Should be exactly same, otherwise 'ssl' might match a service like 'openssl' and 'otherssl'
    if (lc($entryservice) eq lc($matchedservice)) {
        # We have a match on the service
        $service_match = 1;
        debug("'$matchedservice' matches '$entryservice'\n");

        # Proceed to check the all versions if there is any
        if (scalar(@service) > 1) {
      my @versions = split('/',$service[1]);
      # @versions holds the vulnerability DB versions
      # @entryversions holds the host info DB versions
      foreach my $vdb (@versions) {
          foreach my $hdb (@entryversions) {
        # Decide how to compare versions
        # Only supports <1.2.3 and >3.2.1 so far

        # The versioncmp method works just like cmp, but for common version patterns
```

```
382         if ($vdb =~ /([<>])/) {
383             my $meta = $1;
384             if ($meta eq '<') {
385           $vdb =~ s/<//;
386           my $res = versioncmp ($vdb, $hdb);
387           if ($res == 1) {
388               $service_v_match = 1;
389               debug("Version $hdb was smaller than $vdb, version is applicable\n");
390           } elsif (($res == -1) && ($service_v_match == 0)) {
391               $service_v_match = -1;
392               debug("Version $hdb was greater than $vdb, version not applicable\n");
393           }
394             } elsif ($meta eq '>') {
395           $vdb =~ s/>//;
396           my $res = versioncmp ($vdb, $hdb);
397           if ($res == -1) {
398               $service_v_match = 1;
399               debug("Version $hdb was greater than $vdb, version is applicable\n");
400           } elsif (($res == 1) && ($service_v_match == 0)) {
401               $service_v_match = -1;
402               debug( "Version $hdb was greater than $vdb, version not applicable\n");
403           }
404             }
405         } else {
406             # No meta character, just check if versions equal
407             my $res = versioncmp ($vdb, $hdb);
408             $service_v_match = 1 if ($res == 0);
409             debug("Versions $vdb and $hdb are equal\n") if ($res == 0);
410         }
411           }
412       }
413         } else {
414       # Any service version is applicable
415       $service_v_match = 2;
416         }
417     } else {
418         if (($entryservice =~ /\s$/) && ($entryservice)) {
419       # If we get here it means that another service has been detected
420       debug("We were looking for $matchedservice, but the service is actually
              $entryservice\n");
421       $service_match = -1;
422       $service_v_match = -1;
423         }
424     }
425       }
426   }
427     }
428     return ($service_match, $service_v_match);
429 }
430
431 # Fetches all relevant host entries for the current alert
432 # Returns the host information as an array
433 sub getHostInfo {
434     my @info;
435     my $hostinfodsn = "dbi:$hostinfoserver:$hostinfodb:$hostinfohost:$hostinfoport";
436     my $hostinfodbh = DBI->connect($hostinfodsn, $hostinfouser, $hostinfopw) or die "Error
          : Could not connect to the hostinfo database:\n" . $DBI::errstr . "\n";
437
438     my $query;
439     if ($_[1] > 1023) {
```

```
440   # We expect NAT ports above 1023 to be assigned different hosts and services frequently,
          means PRADS data has to 'age' fast
441   my $unixtime = time() - $nat_lifespan;
442   # Get information about the host being targeted in this alert, but only from the last '
          $nat_lifespan' seconds
443   $query = "SELECT IP, META, FIRST_SEEN, LAST_SEEN FROM $hostinfodb WHERE IP='$_[0]' AND
          PORT='$_[1]' AND PROTO='$_[2]' AND LAST_SEEN > 'FROM_UNIXTIME($unixtime)'";
444     } else {
445   # Get information about the host being targeted in this alert
446   $query = "SELECT IP, META, FIRST_SEEN, LAST_SEEN FROM $hostinfodb WHERE IP='$_[0]' AND
          PORT='$_[1]' AND PROTO='$_[2]'";
447     }
448
449     my $result = $hostinfodbh->prepare($query);
450     $result->execute or die "SQL error: $DBI::errstr\n";
451
452     while (my @row = $result->fetchrow_array) {
453         # debug("HostInfo: $row[0], $row[1], $row[2], $row[3]\n");
454   push(@info, $row[1]);
455     }
456
457     $hostinfodbh->disconnect or warn $hostinfodbh->errstr;
458     return @info;
459 }
460
461 # Fetches all relevant vulnerability information for the current alert
462 # Returns the vulnerablity definitions
463 sub getSigInfo {
464     my $os = 0;
465     my $service = 0;
466     my $siginfodsn = "dbi:$siginfoserver:$siginfodb:$siginfohost:$siginfoport";
467     my $siginfodbh = DBI->connect($siginfodsn, $siginfouser, $siginfopw) or die "Error:
          Could not connect to the signature information database:\n" . $DBI::errstr . "\n";
468
469     # Get information about the signature with this ID number
470     my $query = "SELECT os, service FROM vulnerabilityinfo WHERE sig_sid='$_[0]'";
471     my $result = $siginfodbh->prepare($query);
472     $result->execute or die "SQL error: $DBI::errstr\n";
473
474     # Techically this should only ever return one value, since the ID is a unique primary
          key
475     while (my @row = $result->fetchrow_array) {
476         # debug("SigInfo: $row[0], $row[1]\n");
477   $os = $row[0];
478   $service = $row[1];
479     }
480
481     $siginfodbh->disconnect or warn $siginfodbh->errstr;
482     return ($os, $service);
483 }
484
485 # Translates a decimal form IP address to the more familiar dotted version
486 # Returns a dotted format IP address string
487 sub decimalToIp {
488     # join '.', unpack 'C4', pack 'N', shift;
489     inet_ntoa(pack("N", $_[0]));
490 }
491
492 #
493 #####
```

```
494
495 sub usage {
496     # prints the correct use of this script
497     print "This script attempts to validate Snort rules based on\n";
498     print "the available host and vulnerability definitions for\n";
499     print "each alert. Database connection information, NAT lifespan\n";
500     print "and score calculation weights may be adjusted according\n";
501     print "to preferences.\n\n";
502
503     print "Usage:\n";
504     print "-h     Usage\n";
505     print "-v     Verbose\n";
506     print "-d     Debug\n";
507
508     print "./script [-d] [-v] [-h]\n";
509 }
510
511 sub verbose {
512     print "VERBOSE: " . $_[0] if $VERBOSE;
513 }
514
515 sub debug {
516     print "DEBUG: " . $_[0] if $DEBUG;
517 }
```

Code A.1: Prototype for validating alerts with target information

Appendix B

Prototype results

The `sig_priority` column shown in the table below is the priority assigned by Snort while the `status` column contains the score assigned by the prototype.

Table B.1: Prototype results

sig_name	sig_priority	ip_dst	tcp_dport	status
EXPLOIT against IIS 7.5 testrule	1	2150058472	80	100
EXPLOIT against IIS 7.5 testrule	1	2150058472	80	100
EXPLOIT against IIS 7.5 testrule	1	2150058472	80	100
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058474	80	100
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058474	80	100
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058474	80	100
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058474	80	100
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058474	80	100
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
EXPLOIT against Apache < 2.2.22 testrule	1	2150058474	80	80
OpenSSH sshd Identical Blocks DOS attempt	1	2150058289	22	80
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60

Table B.1: Prototype results (continued)

sig_name	sig_priority	ip_dst	tcp_dport	status
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against Apache < 2.2.22 on Linux < 3.0 testrule	1	2150058471	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
EXPLOIT against IIS 7.5 testrule	1	2688863725	80	60
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058413	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058418	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058418	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058251	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058251	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058420	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058420	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058395	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058468	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058468	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058395	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058487	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058404	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058487	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058404	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058413	443	33
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058419	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058419	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058392	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058392	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058403	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058403	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058410	443	10
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058410	443	10
Download of executable content	1	2150058290	55810	0
Download of executable content	1	2150058379	51795	0
Download of executable content	1	2150058331	51300	0
Download of executable content	1	2150058361	53251	0
Download of executable content	1	2150058327	51331	0
Download of executable content	1	2150058322	53169	0
Download of executable content	1	2150058352	49736	0
Download of executable content	1	2150058317	53011	0
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058470	443	0
Download of executable content - x-header	1	2150058336	57483	0
Download of executable content - x-header	1	2150058324	50062	0
DOS Apache mod_ssl non-SSL connection to SSL port	2	2150058494	443	0
Download of executable content - x-header	1	2150058323	59995	0
Download of executable content - x-header	1	2150058322	53178	0
Download of executable content - x-header	1	2150058320	60028	0
Download of executable content	1	2150058370	50847	0
OpenSSH sshd Identical Blocks DOS attempt	1	2150058281	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058375	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058388	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058275	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058389	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058402	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058342	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058330	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058274	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058418	22	-10

Table B.1: Prototype results (continued)

sig_name	sig_priority	ip_dst	tcp_dport	status
OpenSSH sshd Identical Blocks DOS attempt	1	2150058351	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058418	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058312	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058344	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058316	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058314	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058403	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058489	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058349	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058272	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058279	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058386	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058381	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058404	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058348	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058247	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058353	22	-10
OpenSSH sshd Identical Blocks DOS attempt	1	2150058349	22	-10
EXPLOIT against Apache $<$ 2.2.22 on Linux $<$ 3.0 testrule	1	2688863725	80	-120
EXPLOIT against Apache $<$ 2.2.22 on Linux $<$ 3.0 testrule	1	2688863725	80	-120
EXPLOIT against Apache $<$ 2.2.22 on Linux $<$ 3.0 testrule	1	2688863725	80	-120
EXPLOIT against Apache $<$ 2.2.22 on Linux $<$ 3.0 testrule	1	2688863725	80	-120
EXPLOIT against IIS 7.5 testrule	1	2150058474	80	-200
EXPLOIT against IIS 7.5 testrule	1	2150058474	80	-200
EXPLOIT against IIS 7.5 testrule	1	2150058474	80	-200
EXPLOIT against IIS 7.5 testrule	1	2150058474	80	-200